Finding Patterns in Lock-Free Algorithms

Christian Törnqvist
Abstract

Finding Patterns in Lock-Free Algorithms

Christian Törnqvist

Lock-free algorithms are an approach to concurrent programming where threads access shared state without mutual exclusion. Writing correct complex lock-free programs can come with great difficulties. The type system Capable aims to aid the programmer in writing concurrent software, such as lock-free algorithms. This thesis presents an analysis of the current state of Capable and how applicable it is to modern lock-free data structures. It also presents common patterns found in various lock-free data structures, which can be reused when writing new lock-free data structures.
# Table of Contents

1 Introduction .............................................. 4  
   1.1 Overview ............................................. 4  
   1.2 Purpose and Goal ..................................... 4  

2 Background .................................................. 5  
   2.1 Data-Races and Locks ................................. 5  
   2.2 Non-Blocking Algorithms ............................ 6  
      2.2.1 Lock-Freedom .................................... 7  
   2.3 Capable ................................................ 7  
      2.3.1 Capabilities ..................................... 7  
   2.4 Lock-Free Programming in Capable .................. 8  
      2.4.1 Speculation ....................................... 8  
      2.4.2 Linking and Unlinking ............................ 9  

3 Method for Studying Capable ............................ 11  

4 Algorithms .................................................. 12  
   4.1 Treiber Stack .......................................... 12  
      4.1.1 Application of Capabilities ..................... 14  
   4.2 Queue ................................................ 14  
      4.2.1 Description of the Algorithm .................... 14  
      4.2.2 Application of Capabilities ..................... 16  
   4.3 List .................................................. 21  
      4.3.1 Description of the algorithm .................... 21  
      4.3.2 Application of Capabilities ..................... 25  
   4.4 Tree ................................................ 26  
      4.4.1 Description of the algorithm .................... 26  
      4.4.2 Application of Capabilities ..................... 35  
   4.5 Hash Table ........................................... 37  
      4.5.1 Description of the Algorithm .................... 37  
      4.5.2 Application of Capabilities ..................... 46  

5 Validity ..................................................... 47  
   5.1 Unique pointers ....................................... 47  
   5.2 Partial CAT ........................................... 47  
   5.3 Stymied pointers on the heap ........................ 48  

6 Conclusion and Future Work ............................. 49
A Patterns in Lock-free Programs
A.1 Unlinking
A.2 Barring
A.3 Linking

B Algorithms
B.1 Stack
B.1.1 Stack C Implementation
B.1.2 Stack Capable Implementation
B.2 Queue
B.2.1 Queue C Implementation
B.2.2 Queue Capable Implementation
B.3 List
B.3.1 List C Implementation
B.3.2 List Capable Implementation
B.4 Tree
B.4.1 Tree C Implementation
B.4.2 Tree Capable Implementation
B.5 Hash
B.5.1 Hash C Implementation
B.5.2 Hash Capable Implementation
1 Introduction

1.1 Overview

Processor designers have since 2005 begun increasing the number of cores on a single chip to exploit Moore’s law scaling rather than only focusing on single-core performance [1]. To ensure that a single program makes benefit of multiple cores, the programmer needs to write concurrent programs that specifically does this.

Concurrent programming gives rise to problems which do not appear in the code for a single thread. For example, if two threads have shared resources and are allowed to read and write to the same memory location, data-races [2] will arise which results in undefined program behavior [3].

A solution to data-races is to introduce locks. Putting a lock on a shared resource ensures that only one thread can access it at a certain time. However, introducing locks gives rise to other problems. For instance, if a thread cannot retrieve a lock, it can generally not perform any work. Deadlocks [4] can also occur which may leave the system in an unrecoverable state.

To avoid problems like deadlocks, one can choose to implement concurrent algorithms without locks, so called, non-blocking algorithms [5]. In general, an algorithm is said to be non-blocking if a failure or interrupt in one thread cannot cause other threads to also fail. A subset of non-blocking algorithms called lock-free algorithms also guarantee system-wide progress [6].

1.2 Purpose and Goal

This thesis revolves around a type system that is called Capable [7]. The goal of Capable is to support the programmer when writing code for concurrent programs. One way it does this is that it produces errors at compile time if the program is prone to have data-races, for example, if the proper locks are not taken, or if the protocol of a lock-free algorithm is not properly followed.

To design a type system for lock-free algorithms in which relevant algorithms in the literature can be expressed, patterns from such algorithms need to be extracted. Once those patterns are found, they can be used in the development of Capable to assure that Capable is expressive enough to describe those algorithms while still being able to detect data-races.

The core fundamentals of Capable were already described prior to writing this thesis [7]. This project instead revolves around studying in which way Capable needs to be expanded to express different lock-free data structures.
from the literature. These data structures are implemented in C and also in a setting where Capable can be studied.

The goal of this thesis is to find any problems that arise when implementing those algorithms in the Capable type system, and, when possible, propose solutions to these problems.

2 Background

This section explains data-races and disadvantages of using locks, and goes through the concepts of Capable and lock-free programming.

2.1 Data-Races and Locks

A data-race [8] occurs when two or more threads access the same memory location and at least one of the threads executes a write operation and no synchronization operations intervene [2]. Data-races are bugs that can be notoriously difficult to find. A program may continue to execute even after a data-race has occurred making it even more difficult to find the bug. Thus being able to prevent data-races from ever existing in a program, which is one of the purposes with Capable, is very useful.

A data-race can be prevented by putting a lock on a shared resource. Using locks can, however, come with repercussions.

Many advantages of using locks exist. It can for instance often be simple to implement a concurrent algorithm when using mutual exclusion. Programming with locks can, on the other hand, have a negative impact on a program’s performance [9] and introduce various bugs [10].

- **Priority inversion** happens if a lower-priority process is preempted while it has a lock that a higher-priority process needs. This means that the higher-priority process will have to wait for a process that will take longer to finish its work due to having had less CPU time.

- **Deadlock** can happen if different threads were to lock the same objects in different orders.
An example that illustrates this is the figure above. Process A holds Resource B and Process B holds Resource A. When none of the processes are willing to release their resources, a deadlock will occur and the system ends up in a frozen state.

Deadlock avoidance can often be cumbersome if processes are required to lock multiple data structures. This is especially the case if the number of objects that is to be locked is not known in advance.

- Convoying can happen when a process that is holding a lock is descheduled. When this occurs, other processes that require this lock will not be able to progress.

One solution to avoid such problems is to employ lock-free algorithms which fall under the category of non-blocking algorithms.

### 2.2 Non-Blocking Algorithms

Non-blocking algorithms do not allow threads to be blocked \[\text{[11]}\], thereby disallowing locks from being used. Instead of locks, the most common primitive that is used by non-blocking algorithms is called compare-and-swap (CAS). The CAS operation is atomic, and has the following semantics:

```plaintext
CAS(A,B,C) {
    if(A == B) then
        A = C
        return true
    else
        return false
}
```
2.2.1 Lock-Freedom

One highly desired feature of concurrent algorithms is a property that assures system-wide progress in a finite number of steps. All concurrent algorithms do not guarantee this. When one process is continually denied access to the desired resources required to perform its work, it is called starvation.

In non-blocking algorithms, scenarios where all threads in the system are being starved can exist which puts the system in a state where it is unable to make progress. Lock-free algorithms are a subset of non-blocking algorithms that guarantees system-wide progress. Some individual threads may starve indefinitely but the system will still make progress in a finite number of steps.

2.3 Capable

This section describes Capable [7] which is a capability-based static type system for parallel programming. Capable is implemented in a parallel research language called Encore [12]. Encore is not studied in this thesis.

According to the Geneva convention on the treatment of object aliasing [13], alias management schemes can be put into two different categories. These are alias prevention and alias control. In alias prevention systems, compilers and program analyzers can detect during compile-time if there is a possibility for aliasing that leads to a harmful state. The alias control scheme is applied during the run time of a program where it prevents the system from reaching undesirable states.

Capable mainly takes the approach of alias prevention. This means that all checks for any illegal use of aliases are performed at compile time.

2.3.1 Capabilities

In Capable, all references are defined as capabilities. The words references and capabilities will thus be used interchangeably in this thesis from now on. Capabilities can be seen as tokens which govern access rights to objects. They can be further divided into two different categories: non-exclusive and exclusive. Exclusive capabilities are required to be treated linearly. For Capable, this means an exclusive capability is the only reference to the governed object. A consequence to this is that objects protected by exclusive capabilities are safe from data-races.

Non-exclusive capabilities can further be divided into safe and unsafe capabilities. Unsafe capabilities are simply described as capabilities which can be subject to data-races. Capabilities that fall into the safe category are required to follow certain protocols to be thread-safe. Locked and transac-
tional capabilities are classified as safe. Albeit not specified how locked and transactional capabilities are checked to be safe in practice, it is interesting to reason on how this could be achieved. For instance, a way to assure that a locked capability is safe is to require for all of its methods to be wrapped in a Java style “synchronized” block. Regarding the transactional approach, all method calls could be performed inside transactions.

Another safe capability is the Lock-free one which is the main focus of this thesis. An approach on how to assure that lock-free capabilities are “safe” is described in Section 2.4.

The hierarchy for capabilities is displayed in Figure 1.

```
capability
  exclusive
    safe
    lock-free
  non-exclusive
    unsafe
    locked
    transactional
```

Figure 1: different capabilities of Capable

2.4 Lock-Free Programming in Capable

One part of Capable is intended to work for lock-free programming. The idea is to turn exclusive capabilities into safe capabilities with the use of lock-free programming idioms. Achieving this is done by reasoning about ownership of objects, atomic publishing of changes, and speculation.

2.4.1 Speculation

All references on the heap are exclusive when doing lock-free programming in Capable. Fields of an exclusive capability can be marked as speculatable which allows multiple threads to read those fields. The result of reading a speculatable field is a stymied pointer which may only exist on the current thread’s stack. Non-speculatable fields can only be read if the thread has ownership of the object holding that field. This is because threads that own
an object may also write to its non-speculatable fields. If other threads would be able to read from that field, data-races would arise.

### 2.4.2 Linking and Unlinking

A thread may gain ownership of an object by performing **unlinking**. As seen in Figure 2, Node b is a part of the data structure on the left. When unlinking Node b, that node will no longer be reachable within the data structure and Thread 1 has asserted ownership of it which is shown on the right side of the figure. Thread 1 may also insert an object into the data structure, this is called **linking** and is illustrated going from right to left in the figure.

![Figure 2: Results of unlinking and linking](image)

To perform unlinking or linking, a *compare-and-transfer (CAT)* operation is required. A CAT operation is implemented with compare-and-swap but it also serves as a mechanism to transfer ownership of capabilities. Also, when linking an object into a data structure, **CAT_link** is used. This operation is similar to a normal CAT except that the source value is nullified after the CAT succeeds, making it look like the following:

```c
CAT_link(A, B, C) {
    if (CAT(A, B, C)) {
        C = NULL;
        return 1;
    } else {
        return 0;
    }
}
```

Note that the nullification is not atomic. It is, however, safe since it is required for the C-value to only be accessible to the thread performing the CAT operation.

The Treiber stack [14] is one of the simplest lock-free data structures in the literature. Here follows the code of performing a CAT which unlinks an object from the data structure and at the same time, asserts ownership of the unlinked object:
```c
int pop(stack *stack) {
    node *oldTop = speculate stack->top;
    do {
        oldTop = stack->top;
    } while(CAT(stack->top, oldTop, oldTop->next) == 0); //oldTop is not stymied anymore, value can be read
    // (value is not speculatable)
    return oldTop->value;
}
```

Figure 3: Code example of unlinking in a Treiber stack

On line 2, stack->top is speculated on and the result of this speculation is put into oldTop. A while-loop is then entered on lines 3-5, where stack->top is attempted to be replaced by oldTop->next. Once the CAT is completed after line 5, oldTop is not stymied anymore and its value may be read. That is, the ownership of oldTop have been transferred to the thread that completed the CAT.

Below is an example of performing a linking operation in the Treiber stack:

```c
int push(stack *stack, int val) {
    node *newTop = new node;
    newTop->value = val;
    do {
        newTop->next = stack->top;
    } while(CAT_link(stack->top, newTop->next, newTop) == 0);
    return 1;
}
```

Figure 4: Code example of linking in a Treiber stack

An attempt is made to push newTop as a new top. If the CAT succeeds, newTop will have been successfully linked and must not be accessed by the pushing thread. This is achieved by using CAT_link which nullifies the variable after a successful CAT on line 6.
3 Method for Studying Capable

This section outlines a strategy for how to study lock-free capabilities. Five different algorithms are implemented in the programming language C. The reason C is used is due to its versatility and it being a low-level language. Bit manipulation is required for some algorithms and this is easily achieved in C.

Following their implementation, the algorithms were also implemented in a setting where Capable could be studied. This setting is also in C and involves type casting for speculation and macros for CAT operations.

Here is an example of how speculating on an exclusive pointer may look like where Q->head has the type node:

```c
stym_node *head = speculate(Q->head);
```

The speculate function is simply a type cast:

```c
stym_node *speculate(node *n) {
    return (stym_node*) n;
}
```

Compare-and-transfers are implemented using C macros. Here follows an example of such a macro:

```c
#define CAT_link(a,b,c)
(CAS(&a,(node*) b, (node*) c) ? c = NULL, 1 : 0)
```

This macro performs a normal compare-and-swap and if it succeeds, the value of c is nullified and 1 is returned. Otherwise, 0 is returned.

Some algorithms require swapping the least significant bit from 0 to 1, which is called marking. In the original algorithms this is done with a CAS operation. As specified in the Capable article [7], all speculatable fields have mark() operations. A mark function is also provided in this environment and they are implemented as compare-and-swap operations. The reason why this is done with a CAS operation and not a CAT is because no exclusive capabilities from the thread performing the mark are transferred. Therefore, no nullification of the pointer being swapped is needed.
4 Algorithms

This section describes several lock-free algorithms from the literature. Those algorithms include the Treiber Stack [14], a queue [15], a list [9], a binary search tree [16], and a hash table [17]. After the description of each algorithm, an analysis of how well Capable could be incorporated with that algorithm is provided, as well as any issues surfacing when performing this analysis will also be described.

4.1 Treiber Stack

This section describes the Treiber Stack [14], which was briefly displayed in Section 2.4.2. The structs of this data structure are shown in Figure 5. The Stack struct serves as an entrance to the data structure. It contains a field Top which has the type Node.

```
Stack {
    Node *Top;
}
Node {
    int value;
    Node *next;
}
```

Figure 5: Structs of the Treiber Stack

An empty stack is shown in Figure 6. Here, the Top is a null-pointer.

Figure 6: An empty stack
When **pushing** to the stack, a new node, **newTop** is allocated. **newTop**’s **next**-pointer is then set to the node pointed to by the **top**-pointer. This is exemplified in Figure 8 where node 42 is a newly allocated node.

A **CAS** then attempts to swap the **top**-pointer so it points to the new node. In the **CAS**, the **top**-pointer is compared to the **next**-pointer of the new node. If they are equal, the **CAS** succeeds. If it fails, another thread has successfully performed an operation before the compare-and-swap could be finished. This **CAS** is retried until it succeeds and the new node is inserted into the data structure.

If a **pop** is attempted, the current **top** is aliased to a variable called **oldTop** as shown in Figure 9.

A **CAS** is then performed to swap the stack’s **top**-pointer with **oldTop**->**next**. If this succeeds, **oldTop** is unlinked from the data structure and the value can thereafter be safely returned.
4.1.1 Application ofCapabilities

One issue that was found was when creating a new object in the \texttt{push}-function. After \texttt{newTop}'s creation, an assignment to its \texttt{next}-field is performed. The Capable article mentions that a thread can gain exclusive access to a certain capability \[7\]. This exclusivity does however not equal the exclusivity a thread has over an object it has recently created. For instance, in the following \texttt{CAT}, if a thread performing the \texttt{CAT} succeeds, that thread will have exclusive access to \texttt{B}:

\begin{verbatim}
CAT(A,B,C);
\end{verbatim}

This thread is however \textit{not allowed} to do the following if \texttt{next} is speculatable and \texttt{T} is an exclusive pointer on the heap:

\begin{verbatim}
B.next = speculate T;
\end{verbatim}

This is because prior to the \texttt{CAT}, \texttt{A} and \texttt{B} are aliases and another thread may read \texttt{A.next} which would result in a data-race if another thread would be able to write to that memory location.

The following is on the other hand allowed:

\begin{verbatim}
N = new Node;
N.next = speculate T;
\end{verbatim}

This suggests that there are two different types of “exclusivities”.

A proposed name for the type of a newly created object is \textit{Unique}. A unique object may have \textit{all} its fields written to. Once a unique object has been successfully linked into a data structure, it is no longer unique since it may be aliased by several threads.

4.2 Queue

This section describes a lock-free queue algorithm \[15\] written by Michael L. Scott and Maged M. Michael.

4.2.1 Description of the Algorithm

The algorithm is implemented as a singly-linked list with \texttt{Head} and \texttt{Tail} pointers. The \texttt{Head} will always point to a \textit{dummy node} whose \texttt{next}-pointer will always point to the first node in the queue unless the queue is empty, then it is a null pointer. Figure \[10\] shows how an empty queue looks.

When \texttt{enqueueing} a new node \texttt{n} to the queue, the algorithm will attempt to change the \texttt{next}-pointer of the last node \texttt{n_{last}} to point to \texttt{n}. Swapping the
value of $n_{last}$’s `next-pointer` from `null` to $n$ is done with another CAS operation where it is asserted that $n_{last}$ is the last node by checking that $n_{last} \rightarrow next$ is null. If the CAS operation fails, another thread has already enqueued a new node so $n_{last} \rightarrow next$ is no longer null. If the CAS succeeds, the new node is inserted into the queue. The Tail-pointer is however still pointing to the node that was the last node before the insertion, thus, the queue will look like in Figure 11 after the first CAS has succeeded.

An attempt is thus made to `swing` the Tail-pointer to the newly inserted node with a CAS operation. Figure 12 shows how the queue will look once this CAS succeeds. Two CAS-operations has succeeded at this point. The CAS inserting the new node and the CAS swinging the Tail-pointer to the newly inserted node. Once the first CAS is done, the insertion is completed. Because of this, only one attempt is done to swing the Tail-pointer to the newly inserted node. If this CAS fails, no retry is performed. If another thread notices that the Tail is behind, this thread will also attempt to make the same CAS operation that made the Tail pointing to the correct node.
When dequeueing, the first node, $n_{first}$, should be turned into the new dummy node. Before an attempt is made to make $n_{first}$ the dummy node, $n_{first} \rightarrow value$ is stored. After this, an attempt is made to compare-and-swap the Head-pointer to point to $n_{first}$, which is stored in the next-field of the current dummy node. If the CAS succeeds and $n_{first}$ becomes the dummy node, the value of this node has already been stored and the operation is successful. If the CAS fails, the operation is retried from the beginning.

4.2.2 Application of Capabilities

This section describes issues that arise when implementing the algorithm in a Capable setting. Possible solutions to those problems are also discussed.

1. **Problem:** Once the enqueue CAT succeeds, the node that was inserted, newNode, will be nullified in order to disallow the inserting thread from having exclusive access to an exclusive capability on the heap. In the original implementation, once the first CAS operation has succeeded, an attempt is made to compare-and-swap the Tail-pointer so that it points to newNode.

   **Solution:** Since newNode becomes nullified in the Capable version, setting the Tail-pointer to newNode with a compare-and-transfer is not possible. One possible solution to this is to assign the Tail pointer to its next-pointer (which is an alias of the newly nullified node). This alteration does not affect performance since the CAT only moves Tail forward one node which is done in the original algorithm too.

   Another solution that keeps the implementation closer to the original would be if the exclusive reference being transferred would become stymied once the CAT has succeeded instead of nullifying it. This way,
the thread is not able to perform any data-race prone operations on an object that has already been transferred to the heap.

2. **Problem:** When a dequeue is performed, the value of the Head's next-pointer will be compare-and-transferred into the Head. The value-field should not be marked for speculation since a thread that unlinks an object may want to write to value. A thread dequeuing can therefore not read the value of the dummy’s next-pointer before the CAT since it is not speculatable. Neither, can it read the value after the CAT since the object holding the value is nullified in the CAT.

**Solution:** To solve this, the notion of a partial CAT is introduced. If there is an object that has multiple fields, a partial CAT will only transfer a capability to the fields that the object contains.

To denote a partial capability, we write $x \mid f$ to mean “$x$ without the capability to access $f$”. We say that $f$ is barred from $x$. Below is a code example where barring is used. A Node has one int-pointer and one next-field. On line line 6, a new Node $n$ is allocated. On line 7, barring is used. Here, the next-field is barred from the right-hand expression, meaning that it is not included in the assignment. Because of this, the assignment is allowed.
By combining barring with CAT, we get a partial CAT. Using the dequeue operation in the queue data structure as a demonstration, a partial CAT could look like the following:

```c
if(CAT(Q->head, head, head_next | value)) {
    return head_next->value
}
```

Here, value is barred from the compare-and-transfer resulting in that only the next-field is transferred. Applying the partial CAT operation to the queue data structure is sound since the dequeued node will only end up as a dummy node, meaning that the only field of this node that will be of interest is its next-pointer. If the CAT succeeds, the value will no longer be reachable in the data structure resulting in a successful unlinking of value. The thread that performed the CAT has thus acquired ownership of head_next->value and is allowed to write to it.

Another solution is that instead of having partial CATs, value could be marked as speculatable. If this is the case, the thread dequeuing will not have exclusive access to the dequeued object. This can be an issue for programs where complex objects are stored in the queue and where threads are required to perform writes on the fields of objects while they are not in the queue.

3. **Problem:** According to the Capable article [7], Exclusive capabilities must be treated linearly in the program. This means that in order to avoid data-races, there can only be one copy of each exclusive capability in the whole program.

In this algorithm, the Tail-pointer, which is treated as exclusive, will point to one of the last nodes. The problem here is that there will
always be another exclusive pointer that points to the node that Tail points to. Exemplified in Figure 12, Tail points to the same node as the dummy node does. For this algorithm, this is not an issue that will give rise to data-races. There are other algorithms where this could be a problem. Figure 14 displays a data-race prone data structure with multiple exclusive pointers to the same object. If thread 1 owns Alias 1 and thread 2 owns Alias 2, both threads could succeed with partial CATs that would give them exclusive access to Value. Since the threads can now write to the value without any synchronization mechanisms, a data-race has arisen.

Solution: In algorithms where multiple pointers to the same object exist on the heap, it should be allowed for having stymied pointers on the heap. In the example of Figure 14, if the exclusive pointers would instead be stymied, a thread could never gain exclusive access to Value and data-races would not be possible. That is, a thread that would perform unlinking of Node 1 would unlink a node that was already stymied and could thereby not perform any data-race prone operations.

Figure 14: Node 1 has two exclusive pointers to it

4. Problem: When creating the queue, something similar to the following code is required:

```java
    Queue q = new Queue();
    Node head = new Node();
    Node tail = new Node();
    q.head = consume head;
    q.tail = consume tail;
    head.next = tail;
```

Since tail is nullified on line 5, the last assignment is impossible. A solution would instead be to have code looking like the following:
Here is another example of where it is required to have stymied pointers on the heap. As seen, `q.tail` is required to be of a stymied type.
4.3 List

This section describes and analyzes, a non-blocking list algorithm. The data structure was first presented by written by Timothy L. Harris [9].

4.3.1 Description of the algorithm

The list is implemented as a sorted list with keys from a totally ordered universe. It contains two sentinel nodes, one Head, and one Tail node. The sentinel nodes only serve as help nodes and will not contain any inserted key values. Just as in a regular list, each node in the data structure contains two fields, a next-field, and a key-field. The data structure has three available operations, find, insert and delete. Figure 15 and 16 respectively show how an empty and non-empty list looks like.

![Figure 15: An empty list.](image1)

![Figure 16: A list with the values 1 and 2 inserted.](image2)

The algorithm revolves around a private function, not available to the programmer, called search. This function is given a key and returns one left_node and one right_node. The right_node contains the key if it is found by the thread that is calling search. The left_node contains a next-pointer that is pointing to right_node unless the nodes between left_node and right_node has been logically deleted, which is explained in more detail later. If the key is not found, the right_node is the first node that has a key-value greater than key. If no node with a key value greater than key is found, right_node aliases Tail.

The find operation is trivial. The search function is called and a check is made whether right_node.key == key. If this is true, find returns true, otherwise false.
The insertion is also straightforward unless there is contention on the list. As displayed in Figure 17, there is a list of nodes that contains the keys 1, 3, and a new node with the key 2 is to be inserted. The new node has its next-field pointing to the node containing 3. In Figure 18, a CAS operation is attempted on the next-field (red arrow) of the node with the value 1 to swap it with the pointer that is represented as a red dashed arrow. If the CAS succeeds, the resulting list will look like the one in Figure 19.

![Diagram]

Figure 17: An insertion in progress where a new node with the value 2 is allocated.

![Diagram]

Figure 18: Insertion of node 2 in progress. Here, the red dashed line represents the pointer to the newly allocated node 2.
The delete operation is more difficult to perform. Here, it will not suffice with only one CAS operation. In Figure 20, a naive deletion of node 1 is in progress and the next-pointer of the Head has been swapped to node 2. One problem that arises here is that if a thread would want to insert a node with the value 2 concurrently as the deletion was being done, it is possible that it inserts a node with value 2 in between node 1 and 3 even though node 1 has already been deleted. This means that this thread will believe that node 2 was successfully inserted although it is unreachable within the data structure as displayed in Figure 21. To resolve this, the least significant bit of the next-pointer pointing to the node that is being deleted is set to 1. Timothy L. Harris refers to this process as marking [9].

If the previous example is attempted again, this time with marking, before swapping the next-field of the Head node to node 3, the next-field in node 1 is marked. This is displayed in Figure 22.
Now, if the other thread that is trying to insert a node with key value 2, this node would never be inserted in node 1’s next-pointer if it would have been marked. Either, the CAS would fail since the next-pointer has been altered, or the search-function would not return node 1 since it had been marked. Figure [23] also displays how the list looks once node 1 has been physically deleted.

Figure 22: Node 1 is marked for deletion and is logically deleted.

Figure 23: Result of a successful delete operation in a thread-safe implementation. Here, node 1 is physically deleted.

With the concept of marking introduced, search can be explained in more detail. In Figure [24] a list with its first elements, 1, 2, 3 and 4 is displayed. Node 2 and 3 has been logically deleted, but not physically deleted. Calling search with the key 2, 3 or 4 would give a left_node referencing to node 1 and right_node referencing node 4.

Before returning two nodes, search will always assert that left_node and right_node are adjacent. If they are not and there are only marked nodes in between them, search will make them adjacent with a CAS. After calling search on the list in Figure [24] the list will look like the one in Figure [25].

This means that if a successful insertion of the key 2 would be performed, the resulting list would look like the one in Figure [26].

Figure 24: List where node 2 and 3 has been logically deleted
4.3.2 Application of Capabilities

This section discusses, problems and solutions regarding implementing this algorithm in Capable.

1. When physically removing a node in the original search-function, it was possible to return the correct left_node and right_node directly.

   **Problem:** To perform a physical deletion of the nodes in between left_node and right_node, a CAT is required which will compare-and-transfer right_node to left_node->next. Since this will nullify right_node, this node cannot be returned.

   **Solution:** Enter the search-loop again and continue doing this until no physical deletion is required before returning.
2. **Problem:** Just as in the queue data structure, the issue of having multiple exclusive pointers to a single object exists. As displayed in Figure 27, there are multiple exclusive pointers to Tail (List->Tail and Head->next) which is disallowed in Capable.

**Solution:** Similarly to the queue data structure, allowing for pointers on the heap is a valid solution for this data structure, as there is no extraction of values/objects. Both key and next are required to be speculatable in order for this algorithm to work, so having exclusive pointers in the first place is superfluous; since a thread can never get exclusive access to either field anyway. One proposal that builds on this discovery is to *never allow exclusive capabilities to govern objects where all fields are speculatable*. This prevents programmers from creating exclusive capabilities where no thread can get exclusive access to its fields.

### 4.4 Tree

The tree data structure is a binary search tree introduced by Faith Ellen et al. [16].

#### 4.4.1 Description of the algorithm

Just like the list data structure described in this thesis, this BST only works for keys that come from a totally ordered universe. Also similarly to the list, it provides a *find*, *insert*, *delete* operation, and also use a private *search* function.

The BST itself is leaf-oriented, meaning that all nodes that contain the actual keys are located in the leaf nodes. All non-leaf nodes, *internal nodes* will have exactly two children. An invariant is that the left child of node x and all its descendants have values that are strictly less than the value of node x. Also, the right node and all its descendants have values that are strictly greater or equal to the value of node x. The tree also only allows unique keys, so an insertion that would attempt to insert a key that already exists in the tree will return *false*.

The algorithm is non-blocking, meaning that one thread will always make progress in a finite amount of steps. One feature that contributes to this is that different threads will help other threads finish. For example, if thread $t_1$ attempts to perform an insertion and another thread $t_2$ performs an operation that collides with $t_1$’s insertion, $t_2$ will help $t_1$ complete the insertion.

In this implementation, each node maintains pointers to its children but not to its parent. Figure 28 and 29 shows how insertions and deletions
can look like in a BST that is made for one single thread. Inner nodes are represented as circles and leaf nodes as squares.

In Figure 28, an insertion is performed. The leaf node 3 is replaced by a sub-tree, containing one inner node (parent) and two new leaf nodes. The parent will take the max value of the old leaf node and the new node that is to be inserted (2 and 3 in this example). The two leaf nodes will then contain the value of the old leaf node and the newly inserted node.

![Diagram showing insertion](image)

Figure 28: The result of inserting node 2

Figure 29 the deletion of node 2 from the same tree. If the actual node is deleted and nothing more, the constraint that every inner node should have two children would not hold anymore. Instead, the grandparent (node 1) of the β-subtree becomes the new parent of the β-subtree. This is safe because the β-subtree contains the leaf node 3.

![Diagram showing deletion](image)

Figure 29: The result of deleting node 2
For both insertion and deletion, only one single child pointer needs to be changed, and a single CAS operation would be enough if no concurrent updates were present in the system. In the case of concurrent updates, problems arise.

Consider the tree seen in Figure 30 (a). If a Delete(5) and a concurrent Delete(3) execute their CAS operations right after each other, the resulting tree can look like the one in Figure 30 (b). The right child pointer of node 2 (blue arrow) is a result of Delete(3) and the right child pointer of node 4 (green arrow) is a result of Delete(5). Here, node 5 is still wrongfully reachable.

Also, concurrent insertions and deletions can result in a tree that is not possible if all actions were performed in a non-concurrent way. If a concurrent Delete(5) and an Insert(6) would be performed on the tree in Figure 30 (a), one possible tree is shown in Figure 30 (c). The operation Insert(6) creates a new subtree, the one in Figure 30 (c) with a 6 as root. This subtree is inserted as the left child of node 7. Delete(5) then replaces the right child pointer of node 4 with the leaf node 8. This causes the newly inserted node 5 to be unreachable from the root.

Similarly to how the list in Section 4.3 solves interleaving insertions and deletions, this BST also performs marking to notify if a node is in the process of being deleted. In the list, by marking the next-pointer of the node being deleted, no nodes will be inserted after the logically deleted node; that is, once the node is marked, its next-pointer will never change. Similarly, once an inner node has been marked for deletion, it is ensured that none of its child pointers will change.

One problem here is that unlike in the Harris list, the nodes that need
to be protected are not stored in a single word (a node to be deleted has
two children) so it is not possible to atomically mark them with a single
CAS. An Info Record is introduced which contains information about an
ongoing delete or insert-operation. An Info record has all the information
required for another thread to complete the operation. The Info Record
may contain different flags. For instance, if the node is to be deleted its state
is set to MARKED. The Info Record can also have two additional states, Iflag
and DFlag. These two states serve as flags to indicate that an insertion or
deletion is in progress. When an operation finishes, the Iflag or DFlag will
change to CLEAN. The flags essentially serve as locks on the child pointers.
Once the Info Record is flagged, a thread that is attempting to perform a
different operation cannot finish. It is also required that in order for a thread
to change any of the child pointers of a node, it has to first flag it. By doing
this, the problems in Figure 30 (b) and (c) will not arise since if one thread
sees that a node is flagged, it will not change its child pointers.

The insertion will now be done in three CAS-steps. Figure 31 shows an
example where Insert(2) being performed. The following operations are
needed for the insertion and they all require one CAS respectively:

1. Since the leaf node 3 will be replaced with a new subtree, node 3’s
   parent, node 1, is first flagged with an Iflag (shown in Figure 31 (b)).
2. Node 1’s right child is changed to a new sub-tree (shown in Figure 31
   (c)).
3. Node 2’s state is changed to CLEAN (shown in Figure 31 (d)).
In the delete operation, it will not suffice with only three CASes. Instead, four CASes are needed since marking of a node also needs to be performed. The process of performing `Delete(2)` on the tree in Figure 32 (a) is explained below:

1. Node 2’s grandparent 1 is flagged with a DFlag (shown in Figure 32 (b)).
2. Node 2’s parent 3 is set to the state MARKED (shown in Figure 32 (c)).
3. Node 1’s right child is changed to the root node in the β-subtree (shown in Figure 32 (d)).
4. Node 1’s state is changed to CLEAN (shown in Figure 32 (e)).
In this BST, colliding threads will help each other out. The Info Record-pointer points to a struct that contains all information needed to complete an insertion or deletion. In the same word as the Info Record-pointer, the flags IFlag, DFlag, MARKED and CLEAN can be stored. Since they share the same word, this enables flagging and swapping the pointer at the same time.

Figure 33 shows an example of what an insertion can look like in more detail where Info Record’s are included. To clarify the notation of the following figures, pointers that point inside a record will point to that certain record. Pointers that point to the edge of an object points to that object. To the left in the figure, the different fields of the structs are displayed. For instance, the internal node has a State and an Info record-pointer in the same word. It also has a pointer to a left and a right child and also a key. Figure 33 also displays the Info record required for an insertion. It has a pointer to the node’s parent, its parent’s Info record, a pointer to a new
internal node and also a pointer to the leaf node that is to be replaced with the new internal node. Figure 33 (b) exemplifies this. As seen, there is an Info record that points to a future leaf node and also pointing to the leaf that is to be replaced.

1. In (a), the Info Record-pointer has the state CLEAN which is required in order to change its value to start the insert operation.

2. In (b), a new Info Record with an IFlag is allocated. This new Info record has also been compare-and-swapped with the old Info record pointer. This new Info record has pointers to a new internal node (with key 6), the leaf that is to be replaced, the Info record reference of the parent and a pointer to the parent.

3. In (c), the right child of the parent has successfully been compare-and-swapped with the new node.

4. In (d), the iFlag has also been removed with a CAS and the insertion is completed.

Figure 33: A more detailed presentation of how thread-safe insertion can look like.
Figure 34 displays how a deletion looks in a similar fashion. Also shown in this figure is the delete Info Record, which has 5 different fields. One is the Leaf that is to be deleted. Another one is to the Parent which should be marked. The third is the Grandparent where a DFlag will be put. It also contains two pointers to the Parent Info Record which is needed to change this field to MARKED. A Grandparent Info Record-pointer is also stored in order to change it back from DFlag to CLEAN once the deletion has finished.

Here follows the steps required for a scenario where a deletion takes place:

1. (a) displays a sub-part of a tree where the leaf node 3 is to be deleted.
2. In (b), a new Info Record with a DFlag has been compare-and-swapped into the grandparent. The Info record of the grandparent contains, as displayed to the left in the figure, a pointer to the grandparent, a pointer to the grandparent’s Info record, a pointer to the parent, the parent’s Info record and the leaf node.
3. In (c), the parent has been marked for deletion.
4. In (d) the right child of the grandchild has been successfully compare-and-swapped with the child that was not to be deleted. After this, the state of the Info record needs to be changed from DFlag to CLEAN with a CAS in order for the deletion to be completed.

Figure 34: A more detailed presentation of how a thread-safe deletion can look like.
If a thread that is performing an insertion manages to flag the parent with an IFlag, the insertion is guaranteed to succeed. A deletion does, on the other hand, need to succeed with two CASes in order to be guaranteed to succeed. Once the parent has been marked with a MARK, the deletion will be completed in a finite number of steps. If threads attempting the deletion operation would successfully flag the grandparent with a DFlag but failing with the MARK because another thread may have flagged that node, a backtrack is necessary for which the DFlag will be removed. A scenario where a backtrack would be needed is shown in Figure 35. Since the parent of node 4 never got marked but instead flagged with an IFlag, the DFlag needs to be removed and the operation will fail.

Figure 35: Node 2 is successfully flagged with a DFlag but another thread have flagged the inner node 4 with an IFlag. This means that the thread performing the deletion (or any other thread that try to help) will have to perform a backtrack where the DFlag is changed to CLEAN.

A scheme for possible sequences of marks that a single Info record can have is shown in Figure 36. Here, dchild CAS stands for the deletion of its child. ichild CAS stands for an insertion of a new node with a CAS. Also as seen, there is no way for an Info record to get rid of its MARK.
4.4.2 Application of Capabilities

1. **Problem**: As mentioned in the previous section, this algorithm requires a bit-level representation of 4 different states and those states should be stored in the info-record pointer. However, in the Capable article [7], only a `mark()` operation exist.

   **Solution**: There are two possible solutions to this problem. One option is to allow for freely altering the two most least significant bits with a **CAS** operation. Another solution for this particular algorithm is to have *proxy objects*. So instead of having an info-record pointer stored in each node, each node should point to a proxy object. This proxy object should then have a type which could be stored as a field of the object. The proxy would then in turn point to the actual info-record. So for instance, when changing the state of a node from **CLEAN** to **MARKED**, this would not be performed with a bit-manipulation using **CAS**. Instead, a **CAS** would be performed to change the proxy pointer to a proxy object that is marked.

![Figure 36: Possible sequences of the states of a single Info record](image-url)
2. **Problem:** This algorithm also suffers from multiple exclusive pointers on the heap. Just like in the previous data structures, the fields of all structures need to be speculated on. One proposed solution for this particular data structure is thus to only have stymied pointers on the heap and not use exclusive capabilities at all.

If the programmer requires values to be extracted from the data structure, it would suffice only have the child-pointers as exclusive. The unlinking would occur in the function `CAS_child` shown below which replaces a leaf with a new sub-tree (`new_subtree`). After the CAT succeeds, the thread will have unlinked `leaf`.

```c
1 CAT_child(parent, leaf, new_subtree) {
2   if (new_subtree->key < parent->key) {
3     CAT(parent->left, leaf, new_subtree);
4   } else {
5     CAT(parent->right, leaf, new_subtree); }
```
4.5 Hash Table

Hash tables are often used to implementing set and map data structures since hash tables generally provide insertion, deletion, and lookup in constant time. A hash table typically consists of a bucket array. Each bucket is a pointer to a dynamically growing and shrinking set object. A hash function is also used which will, given a key, find the bucket that this key belongs to in constant time. When the number of elements in the hash set increases, the risk of hash collisions also increases. Often a max number of elements per bucket is allowed. Once the number of elements in one bucket grows above the size cap, the number of buckets in the hash table is doubled. Resizing a hash table in a concurrent environment can be difficult. For instance, a new hash function needs to be used in order to make use of the newly allocated buckets. If a new hash function is used, elements that belonged to bucket $i$ before the resize might not belong to bucket $i$ after the resize. Because of this, when resizing, all elements must be moved to their new buckets. Moving multiple objects atomically is typically not something that done easily in a lock-free environment.

To solve this, Michael Spear et al. implements freezable sets [17]. The freezable set $\text{fset}$ will serve as the buckets in the hash set. Freezable sets support a freeze operation which turns the set immutable and returns the final state of the set. This is handled by an ok-field which tells whether the set is mutable or not. $\text{fset}$ supports lookups, insertions, and deletions.

4.5.1 Description of the Algorithm

The main entrance to the data structure is via a pointer called $\text{head}$. Since the hash set can be resized, and doing this will result in the need of allocating new buckets, the $\text{head}$-pointer will always point to the most recent hash set $\text{hnode}$. Figure 37 shows an empty hash set where the $\text{head}$ points to an $\text{hnode}$. As seen in the figure, there are three variables, $\text{fset[]}$, $\text{Size}$ and $\text{Pred}$. 
FSet[] is a bucket and is pointing to an array of FSets. Every FSet, in turn, has a pointer to an FSetNode which holds the keys for this bucket.

Size denotes the number of buckets that can be used in the current HNode.

Pred points to the previous HNode. This means that if a resize has taken place, a new HNode will be allocated, and Pred will point to HNode that Head pointed to before the resize. In this example, no resize has taken place so Pred is a null-pointer.

Figure 37 shows a hash set with two keys, Q and R inserted. It is the first allocated HNode so no preceding Head exist. As seen, the Size is 1 so the size of the bucket array is also 1. The one element in the bucket array points to a structure containing a pointer to a structure which contains the keys 1 and 2. It also has an indication ok == true which tells that the set is not frozen.
Three functions are available to the programmer; Insert, Delete and Contains.

Insert attempts to insert a key into the set. If the key already exists, Insert will return false since no insertion was performed. If the key did not exist, it will return true.

Delete work in a similar fashion as Insert but will instead return true if the key was in the set and false otherwise.

More detail on Insert and Delete is given below. Contains is on the other hand trivial - it simply checks the bucket provided by the hash function if the key is there. If it is not, it will look in the predecessing HNode if the key exists there.

In this algorithm, new buckets are allocated lazily, meaning that after a resize, initially all elements in FSet[] are null. When an insertion or deletion is performed on a certain key and its corresponding bucket is not initialized, a function, initBucket() is called. It is used for allocating a new bucket and migrating the keys from the old bucket(s).

For this first example, assume that no bucket has been allocated in the new HNode and the operation insert(3) is to be performed on the hash structure seen in Figure 39. The process of performing this insertion is explained below and demonstrated in Figure 40. For simplicity, the most recent HNode is referred to as HNodeNew and the old one as HNodeOld.
1. Looking at Figure 40 (a), insert(3) is applied. The hash function key \( \text{mod} \) \( \text{Size} \) is used when looking for a bucket. \( 3 \mod 2 = 1 \) so \( \text{FSet}[1] \) in \( \text{HNodeNew} \) is checked. Since \( \text{FSet}[1] \) is null, \( \text{initBucket()} \) is called. The same hash function is used in \( \text{HNodeOld} \) but here with \( \text{Size} = 1 \). Therefore, \( 3 \mod 1 = 0 \) so the keys from \( \text{FSet}[0] \) in \( \text{HNodeOld} \) should be migrated. Before this is done, the \( \text{FSetNode} \) in \( \text{FSet}[0] \) of \( \text{HNodeOld} \) needs to be frozen so no keys are inserted or deleted during the migration. As seen in Figure 40 (a), the red pointer is to be swapped with the dashed pointer using a CAS operation.

2. In Figure 40 (b), two CAS operations have succeeded. First, \( \text{FSet}[0] \) in \( \text{HNodeOld} \) now points to a new \( \text{FSetNode} \) with \( \text{ok} = \text{false} \). Also in \( \text{HNodeNew} \), \( \text{FSet}[1] \) is now pointing to a bucket which contains the same elements as \( \text{FSet}[0] \) in \( \text{FSetOld} \).

3. In Figure 40 (c), a new \( \text{FSetNode} \) is allocated with a dashed pointer to it. The red pointer should be swapped with the dashed one and this is done with another CAS operation.

4. In Figure 40 (d), the CAS has succeeded and the key 3 has been successfully inserted into the data structure.

A delete operation, e.g. \( \text{Delete}(1) \) would occur in a very similar way. The difference is in Figure 40 (c), where the newly allocated \( \text{FSetNode} \) would instead point to an empty set.
Figure 40: How an insertion can look like when no designated bucket has been allocated for a certain key.

If a downsize has occurred, the number of buckets in the current HNode will be less than the ones in the predeceasing HNode. Since a new hash function is used, this results in that two different keys that would belong to separate buckets in the previous HNode, now may need to share the same one. Figure 41 displays an example where a downsize has occurred. Here follows the process of how the migration of keys would occur if an insertion or deletion were to be performed on any key. For simplicity, say insert(5) is performed

1. The bucket in HNodeNew is null, so initBucket() is called.
2. When initBucket() is called, it performs a check and sees that a downsize has occurred.
3. It will freeze the buckets HNodeOld.FSet[5 mod HNodeNew.size] (= 0) and HNodeOld.FSet[5 mod HNodeNew.size + HNodeNew.size] (= 1).
4. Once both sets are frozen, the union of those sets creates a new set which is set to the new bucket containing the keys 0 and 1 as seen in Figure 42 (a).
5. Figure 42 (b) shows the final state of HNodeNew where the CAS operation has successfully been completed.

42

Figure 41: Hash set where a downsize has occurred.

Figure 42: Buckets in HNodeOld frozen and the progression of key migration

The contains operation is trivial. Given a key, it looks for the bucket in the most recent HNode (HNodeNew in Figure 43). If this bucket is null, it will look in the predecessor. For example, in Figure 43 contains(2) would look in the first bucket in HNodeNew and contains(1) would first look in the
second bucket in HNodeNew but since it was null, it would look in HNodeOld. There, it would find the key 1 and return true.

![Diagram of hash set](image)

Figure 43: Hash set where the key 1 is only reachable through HNodeOld and the key 2 is reachable through HNodeNew.

Finally, here is an explanation of how a resize is handled. There are two different scenarios when performing a resize. The simplest scenario is the first resize that occurs when Pred is a null pointer. The other scenario is when Pred is not null.

First consider the simpler scenario. In Figure 44, the bucket size limit is set to 3, so a resize will begin once the hash structure reaches the state shown in Figure 44(b).
As seen in Figure 45 (a), after the insertion, a new \texttt{HNode} will be allocated. In Figure 45 (b), the \texttt{HNode}-pointer has been successfully been swapped to the new \texttt{HNode} using a \texttt{CAS}.

When performing a resize in the more complicated scenario, it is \textit{not} guaranteed that all buckets in the preceding \texttt{HNode} have been initialized.
Because of this, an initBucket() is performed on each bucket in the HNode that is to be resized. In Figure 46, a resize of HNode1 is in progress. Before allocating a new HNode, all buckets (only one in this example) in HNode1 will have initBucket() applied to them. This is done in Figure 47 (a). In Figure 47 (b), a new HNode has been allocated. Once initBucket() has been called on all buckets, the Head pointer is compare-and-swapped to the most recent one.

Figure 46: Resize where the previous HNode's pred-pointer is not null.

Figure 47: Same structure as in Figure 46 with initBucket() performed in (a) and a new HNode allocated in (b).
4.5.2 Application of Capabilities

No discovery that has not already been discussed in previous algorithms was found for the hash data structure.
5 Validity

In parallel to writing this thesis, the work on Capable has progressed, resulting in a type system called Lolcat [18] (short for "Lock-free Linear Compare and Transfer"). To validate the work behind this project, the key findings of this thesis are in this section compared to similar solutions in Lolcat. Lolcat has been used to implement several of the data-structures from this thesis and has a formal proof of soundness.

5.1 Unique pointers

Section 4.1.1 discussed the need for a unique pointer. When creating a new object, a thread should be able to write to all of its fields without synchronization regardless of if a field is speculatable or not. Therefore, the unique pointer was introduced where a thread is guaranteed to have exclusive access to an object and does thereby not have to worry about any data-races. When the thread creating the unique pointer inserts it into a data structure, it should no longer be unique. In Lolcat, a very similar concept is the type annotation pristine. A reference to an object will have the type annotation pristine if it has just been created and there are no aliases of that object. Similar to unique pointers, pristine ones cease to be “pristine” once they are inserted into a data structure. An object that has ceased to be pristine may also never become pristine again, this also applies to unique pointers as discussed in Section 4.1.1.

5.2 Partial CAT

Another proposal which was discussed in Section 4.2.2 was the partial CAT. The purpose of a partial CAT is to only transfer a subset of the fields of an object. The fields that are not transferred become exclusive to the thread performing the CAT, given that those fields are not speculatable. Partial CATs were used in conjunction with what was referred to as barring and could look like the following:

```
CAT_unlink(Q->head, head, head_next | value)
```

If this operation succeeds, head->value becomes exclusive to the thread performing the CAT.

Lolcat can also perform operations that resemble partial CATs. This is done by using field restrictions in combination with residual aliasing. Three different kinds of field restrictions exist in Lolcat, but for this demonstration, only one needs to be described: strongly restricted types. If a reference \texttt{R} with
the field \( f \) has the strongly restricted type \( R \mid f \), \( f \) is guaranteed to never be accessed again in the whole system.

Here follows an example of when this is used in action:

```java
Node {
    int elem;
    Node next;
}

Queue {
    Node || elem first;
}

def dequeue(Queue q) {
    ... 
    CAT(q.first, oldFirst, oldFirst.next)
}
```

In this listing, `Queue` has one `first` which has the type `Node || elem`. Before the `CAT` on line 12, `oldFirst.next` has exclusive access to `oldFirst.next.elem`. If the `CAT` succeeds, `oldFirst.next` is transferred to `q.first`. Since `q.first` has the type `Node || elem`, the `elem` (previously `oldFirst.next`) cannot be accessed anywhere in the program after the `CAT`. **LOLCAT** provides a way to retrieve this lost `elem` for the thread that performed the `CAT`. This is done with residual aliasing and looks like the following:

```java
CAT(q.first, oldFirst, oldFirst.next) => elem
```

Once this is done, the thread that performed the `CAT` will have exclusive access to `elem`. The result of performing this kind of `CAT` in **LOLCAT** is the same as the partial `CAT` described in Section 4.2.2.

### 5.3 Stymied pointers on the heap

When implementing the algorithms, it was apparent that the constraint of only having exclusive pointers on the heap needed to be broken. A proposed solution to this was to allow for stymied pointers on the heap. This solution essentially means that a thread would be able to speculate on speculatable fields, but it may never access or gain exclusive access to non-speculatable fields. Again, **LOLCAT** solves this by using field restriction, here called *weakly restricted types*. If a reference \( R \) with the field \( f \) has the weakly restricted type \( R \mid f \), \( f \) can never be accessed through \( R \). \( f \) can moreover be accessed via other references in the system, unlike what would be the case if the field were strongly restricted. Even if a thread would gain ownership over \( R \), \( f \) would
not be possible to access. This solution is very similar to the one involving stymied pointers on the heap as discussed in Section 4.2.2.

6 Conclusion and Future Work

This thesis lays a foundation for what could be worked upon and incorporated into Capable in the future. Five different lock-free algorithms have been implemented in C. They have also been implemented in C programs which emulate the fundamentals of Capable.

For some of the algorithms, it was found that they required small modifications in order for them to work in Capable. One drawback in those modifications could involve having additional loops in the process of inserting or deleting an element (for example as observed in the list data structure, Section 4.3) which can cause a drawback on performance. No benchmarks for doing this were however performed, though this is something that should be done in future work.
References


50


A Patterns in Lock-free Programs

One concept that was discussed during the writing of this thesis was that patterns in lock-free programs which could be sugarized should be discovered. If such patterns existed, syntactic sugar on that type of code would improve code readability and maintainability. This section is put in the appendix due to it may be of use to the Capable team, yet not relevant enough to be a part of the thesis.

In many of the algorithms, the code was too ad hoc for it to be convenient to extract some generic pattern. Some common patterns were however found that could be built upon. All algorithms had some deletion and insertion and they could be divided into two groups, linking and unlinking. That is, when a deletion is made, an element is unlinked from the data structure and vice versa.

A.1 Unlinking

Unlinking could for some algorithms be abstracted into a more generic pattern. For the stack and queue data structures, a common pattern was found. It looks similar to the pseudo representation below where one data structure pointer, Q where a field n, is present:

```plaintext
do {
    <Assignment to variables old_n and old_n_next>
    <Operations and conditionals based on old_n, old_n_next and Q>
} while (CAS(Q->n, old_n, old_n_next) == false)
```

Figure 48: Generic patterns extracted from the stack and queue data structures when unlinking

Here follows a suggestion for syntactic sugar which could be applied to that pattern:
As seen in Figure 49 a prime, ’ is used. Instead of assigning to Q->n to old_n as done in Figure 48 referring to what would have been old_n can be done by adding a prime to Q->n, that is Q->n’. Keywords like continue will not work in a similar fashion to how they do in a do-while loop in C. In C, a continue in a do-while loop will execute the statement enclosed by the while. In the sugarized context described above, a continue statement will restart the loop without attempting to perform the actual CAT.

When a new round of the loop is run, all assignments in the let statement will be remade. Reassignments to all primed pointers will also be performed, not visible to the programmer. A break can also be used for exiting the unlink-procedure.

### A.2 Barring

As discussed in Section 4.2.2 a partial CAT can sometimes be desired. To represent this in this context of syntactic sugar, a bar, ’|’, is used. Below is an example of how this could be represented when unlinking. The pointer Q has two fields, next and value.

```plaintext
if(let
  <Assignment to variables x and y>
  in unlink(Q->n, (Q->n->next) | value)
then return Q->n->next | next
```

In the unlink statement, (Q->n->next) | value is transferred to Q->n. This means that value is barred and is thus not transferred. Because of the if-statement, it is certain that the unlinking succeeded. It is, therefore, safe
to return the field value of Q->n->next since the current thread is the only one that has access to it.

Note that if the return statement would instead return Q->n->next | value, this would result in a syntax error. The reason for this is that the next-value is an exclusive pointer and thus not directly accessible to the current thread.

Barring can also be used anywhere else in the code. For instance in the queue algorithm when enqueuing, the head’s value does never need to be accessed. A CAT like the following could thus be used when dequeuing: CAT(Q->head, Q_head, Q_head_next | value). If a Node would contain the fields next and value, Q->head should have the type Node | value.

A.3 Linking

The patterns found for linking is very similar to unlinking. They were also extracted from the more simple data structures stack and queue.

```c
new_node = new Node
do {
  <Assignment to variables new_node, old_head>
  <Operations and conditionals based on new_node, old_head and Q>
} while(CAS(Q->head, old_head, new_node) == false)
```

Figure 51: Generic patterns extracted from the stack and queue data structures when linking

Here follows a suggestion for syntactic sugar which could be applied to that pattern:

```c
new_node = new Node
let
  <Assignment to variable new_node>
in link(Q->head, new_node)
  <Operations and conditionals based on new_node, Q->head, Q->head’, Q and Q’ etc>
```

Figure 52: Possible syntactic sugar for linking
B Algorithms

B.1 Stack

B.1.1 Stack C Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define CAS(a,b,c) __sync_bool_compare_and_swap(&a,b,c

typedef struct node {
    int value;
    struct node *next;
} node;

typedef struct stack {
    node *top;
} stack;

stack *init() {
    stack *init_stack = malloc(sizeof(stack));
    return init_stack;
}

int pop(stack *stack) {
    node *old_top = stack->top;
    int returnVal = 0;
    do {
        old_top = stack->top;
    } while CAS(stack->top, old_top, old_top->next) == 0);
    returnVal = old_top->value;
    return returnVal;
}

int push(stack *stack, int val) {
    node *newTop = malloc(sizeof(node));
    newTop->value = val;
```
do {
    newTop->next = stack->top;
}
while(CAS(stack->top, newTop->next, newTop) == 0);
return 1;
B.1.2 Stack Capable Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define CAT(a,b,c) (__sync_bool_compare_and_swap(&a,(node*) b, (node*) c) ? c = NULL, 1 : 0)
#define CAT_unlink(a,b,c, barred) (  
__sync_bool_compare_and_swap(&a,(node*) b, c) ?  
barred->value = b->value, c = NULL, 1 : 0)

typedef struct node {
    int value;
    struct node *next;
} node;

typedef struct stym_node {
    int value;
    struct node *next;
} stym_node;

typedef struct stack {
    node *top;
} stack;

typedef struct node_has_int {
    int value;
} node_has_int;

stym_node *spec_top(stack *s) {
    return (stym_node*) s->top;
}

void set_next(stym_node *n, stym_node *new_next) {
    n->next = (node*) new_next;
}

stack *init() {
    stack *init_stack = malloc(sizeof(stack));
    return init_stack;
}
```

57
37 }
38 int pop(stack *stack) {
39     stym_node *oldTop = spec_top(stack);
40     int returnVal = 0;
41     while(1) {
42         oldTop = spec_top(stack);
43         if(oldTop != NULL) {
44             if(CAT(stack->top, oldTop, oldTop->next)) {
45                 break;
46             }
47         }
48     }
49     returnVal = oldTop->value;
50     return returnVal;
51 }
52
53 int push(stack *stack, int val) {
54     stym_node *newTop = malloc(sizeof(node));
55     stym_node *oldTop = spec_top(stack);
56     newTop->value = val;
57     do {
58         oldTop = spec_top(stack);
59         set_next(newTop, oldTop);
60     } while(CAT(stack->top, newTop->next, newTop) == 0);
61     return 1;
62 }
B.2 Queue

B.2.1 Queue C Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define CAS(a,b,c) __sync_bool_compare_and_swap(&a,b,c

typedef struct node {
    int value;
    struct node *next;
} node;

typedef struct queue {
    node *head;
    node *tail;
} queue;

int init(queue *queue) {
    node *init_node = malloc(sizeof(node));
    init_node->value = 0;
    init_node->next = NULL;
    queue->head = queue->tail = init_node;
    return 1;
}

int enqueue(queue *Q, int val) {
    node *new_node = malloc(sizeof(node));
    new_node->value = val;
    new_node->next = NULL;
    node *tail = Q->tail;
    node *tail_next = tail->next;
    while (1) {
        tail = Q->tail;
        tail_next = tail->next;
        if (tail == Q->tail) {
```
if (tail_next == NULL) {
    if (CAS(tail->next, tail_next, new_node)) {
        break;
    } else {
        CAS(Q->tail, tail, tail_next); // tail behind
    }
}

CAS(Q->tail, tail, new_node); // tail behind
return 1;

int dequeue(queue *Q, int *pvalue) {
    node *head = Q->head;
    while(1) {
        head = Q->head;
        node *tail = Q->tail;
        node *head_next = head->next;
        if (head == Q->head) {
            if (head_next == NULL) { // Is queue empty or Tail falling behind?
                return 0; //Queue is empty
            } else {
                *pvalue = head_next->value;
                if (CAS(Q->head, head, head_next)) {
                    break;
                }
            }
        }
        free(head);
        return 1;
    }
B.2.2 Queue Capable Implementation

```c
#include <unistd.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define CAT(a,b,c) (__sync_bool_compare_and_swap(&a,(node*) b, (node*) c) ? c = NULL, 1 : 0)
#define CAT_unlink(a,b,c, barred) ( __sync_bool_compare_and_swap(&a,(node*) b, (node*) c) ? barred->value = c->value, c = NULL, 1 : 0)

typedef struct node {
    int value;
    struct node *next;
} node;

typedef struct node_val {
    int value;
} node_val;

typedef struct stym_node {
    int value;
    struct node *next;
} stym_node;

typedef struct link_node {
    struct node *node;
} link_node;

typedef struct queue {
    node *head;
    node *tail;
} queue;

typedef struct node_has_int {
    int value;
} node_has_int;
```
`stym_node *speculate(stym_node *n) {
    return (stym_node*) n;
}

stym_node *get_next(stym_node *n) {
    return (stym_node*) n->next;
}

queue *init() {
    queue *Q = malloc(sizeof(queue));
    node *init_node = malloc(sizeof(node));
    init_node->value = 1;
    init_node->next = NULL;
    Q->head = Q->tail = init_node;
    return Q;
}

int enqueue(queue *Q, int val) {
    node *new_node = malloc(sizeof(node));
    new_node->value = val;
    new_node->next = NULL;
    stym_node *tail = speculate(Q->tail);
    stym_node *next = (stym_node*) tail->next;

    while (1) {
        tail = speculate(Q->tail);
        next = (stym_node*) tail->next;
        if (tail == speculate(Q->tail)) {
            if (next == NULL) {
                if (CAT(tail->next, next, new_node)) {
                    break;
                }
            } else {
                CAT(Q->tail, tail, next);
            }
        } else {
            node *tail_next = tail->next;
            CAT(Q->tail, tail, tail_next);
            return 1;
        }
    }
}
```c
int dequeue(queue *Q) {
    stym_node *head = speculate(Q->head);
    stym_node *tail = speculate(Q->tail);
    stym_node *next = speculate(head->next);
    node_has_int *barred_value = malloc(sizeof(
        node_has_int));
    int return_value = 0;

    while(1) {
        head = speculate(Q->head);
        tail = speculate(Q->tail);
        next = speculate(head->next);
        if (head == speculate(Q->head)) {
            if (head == tail) { // Is queue empty or Tail
                if (next == NULL) {
                    return 0; //Queue is empty
                }
                CAT(Q->tail, tail, next);
            } else {
                if(CAT_unlink(Q->head, head, next, barred_value)) {
                    return_value = barred_value->value;
                    return return_value;
                }
            }
        }
    }
    return 1; // not reached
}
```
B.3 List

B.3.1 List C Implementation

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

typedef struct node {
    int key;
    struct node *next;
} node;

typedef struct list {
    node *head;
    node *tail;
} list;

node *search(list *list, int search_key, node **left_node);

intptr_t is_marked(node *node) {
    int ret = ((intptr_t) node >> (sizeof(intptr_t)*8 - 1)) & 1;
    return ret;
}

intptr_t get_marked_reference(node *node) {
    intptr_t temp = (intptr_t) node;
    temp |= (intptr_t) 1 << (intptr_t) (sizeof(intptr_t) *8 - 1);
    return temp;
}

intptr_t get_unmarked_reference(node *node) {
    intptr_t temp = (intptr_t) node;
    temp &= ~(intptr_t) 1 << (intptr_t) (sizeof(
        intptr_t)*8 - 1));
    return temp;
```

64
```c
34 }
35
36 list *init_list() {
37     node *head = malloc(sizeof(node));
38     node *tail = malloc(sizeof(node));
39     head->next = tail;
40     list *new_list = malloc(sizeof(list));
41     new_list->head = head;
42     new_list->tail = tail;
43     head->key = -1;
44     tail->key = -2;
45     return new_list;
46 }
47
48 int insert(list *list, int key) {
49     node *new_node = malloc(sizeof(node));
50     new_node->key = key;
51     node *right_node = list->head;
52     node *left_node_ref, *left_node;
53     do {
54         right_node = search(list, key, &left_node_ref);
55         left_node = (node*) left_node_ref;
56         if ((right_node != list->tail) && (right_node->key
57             == key)) { //Only unique values
58             return 1;
59         }
60     } while (__sync_bool_compare_and_swap(&left_node->
61             next, right_node, new_node));
62     return 1;
63 }
64
65 int delete(list *list, int search_key) {
66     node *right_node, *right_node_next, *left_node, *
67         left_node_ref;
68     do {
69         right_node = search(list, search_key, &
70             left_node_ref);
71         left_node = (node*) left_node_ref;
72         if ((right_node == list->tail) || (right_node->key
73             != search_key))
```
//right_node is a dummy node or doesn’t exist in the list
return Ø; // Deletion failed but element not in list

right_node_next = right_node->next;
if (!is_marked(right_node_next)) {
  // Logical deletion
  if (__sync_bool_compare_and_swap(&(right_node->next),
    right_node_next, get_marked_reference(right_node_next)) {
    break;
  }
}
} while (1);

// Physical deletion
if (!(__sync_bool_compare_and_swap(&(left_node->next),
    right_node, right_node_next))) {
  right_node = search(list, right_node->key, &
    left_node_ref);// left_node);
}
return 1;

int find(list *list, int search_key) {
  node *right_node, *left_node_ref;
  right_node = search(list, search_key, &left_node_ref);
  if (((right_node == list->tail) || (right_node->key
    != search_key)) {
    return Ø;
  } else {
    return 1;
  }
}
node *search(list *list, int search_key, node **
left_node) {
  node *left_node_next, *right_node;
search_again:
  while (1) {
    node *t = list->head;
    node *t_next = (list->head)->next;
    
    // 1: Find left_node and right_node
    do {
      if (!is_marked(t_next)) { // If t is not marked
        *left_node = t;
        left_node_next = t_next;
      }
    } while(is_marked(t_next) || (t->key < search_key));
    right_node = t; // right_node = the node being searched for
    
    // 2: Check nodes are adjacent
    if (left_node_next == right_node) {
      if ((right_node != list->tail) && is_marked(right_node->next)) {
        // Search again if right_node is marked and is not the last (dummy) node.
        goto search_again;
      } else {
        return right_node;
      }
    }
    
    // 3: Remove one or more marked nodes
    if (__sync_bool_compare_and_swap(&(*left_node)->next, left_node_next, right_node)) {
      if ((right_node != list->tail) && is_marked(right_node->next)) {
// Search again if right_node is marked and is not the tail node.
goto search_again;
} else {
    return right_node;
}
B.3.2 List Capable Implementation

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#define CAT(a, b, c) (__sync_bool_compare_and_swap(&a, (node*) b, (node*) c) ? c = NULL, 1 : 0)
#define CAS(a, b, c) (__sync_bool_compare_and_swap(&a, (node*) b, (node*) c))

typedef struct node {
    int value;
    struct node *next;
} node;

typedef struct stym_node {
    int value;
    struct node *next;
} stym_node;

typedef struct list {
    node *head;
    node *tail;
} list;

typedef struct node_pair {
    stym_node *left;
    stym_node *right;
} node_pair;

stym_node *speculate(node *n) {
    return (stym_node*) n;
}

stym_node *get_next(stym_node *n) {
    return (stym_node*) n->next;
}
```
void set_next(stym_node *n, stym_node *next) {
    n->next = (node*) next;
}

int get_value(stym_node *n) {
    return n->value;
}

node_pair *search(list *list, int value);

int is_marked(stym_node *node) {
    int ret = ((intptr_t) node >> (sizeof(intptr_t)*8
                 -1)) & 1;
    return ret;
}

intptr_t get_marked_reference(stym_node *node) {
    intptr_t temp = (intptr_t) node;
    temp |= (intptr_t) 1 << (intptr_t) (sizeof(intptr_t)
                          *8 - 1);
    return temp;
}

intptr_t get_unmarked_reference(stym_node *node) {
    intptr_t temp = (intptr_t) node;
    temp &= ~((intptr_t) 1 << (intptr_t) (sizeof(intptr_t)
                             *8 - 1));
    return temp;
}

int mark(node *orig, stym_node *ref) {
    stym_node *marked_ref = (stym_node*)
                             get_marked_reference(ref);
    return CAS(orig, ref, marked_ref);
}

list *init_list() {
    node *head = malloc(sizeof(node));
    node *tail = malloc(sizeof(node));
    head->next = tail;
    list *new_list = malloc(sizeof(list));
}
new_list->head = head;
new_list->tail = tail;
head->value = -1;
tail->value = -2;
return new_list;
}

int insert(list *list, int value) {
    stym_node *new_node = malloc(sizeof(node));
    new_node->value = value;
    stym_node *right_node = speculate(list->head);
    stym_node *left_node;
    node_pair *node_pair;
    do {
        node_pair = search(list, value);
        left_node = node_pair->left;
        right_node = node_pair->right;
        if ((right_node != speculate(list->tail)) && (get_value(right_node) == value)) { //Only unique values
            free(node_pair);
            return 1;
        }
        set_next(new_node, right_node);
        if ((CAT(left_node->next, right_node, new_node))) {
            free(node_pair);
            return 1;
        }
    } while (1);
}

int delete(list *list, int value) {
    stym_node *left_node;
    stym_node *right_node, *right_node_next;
    node_pair *node_pair;
    do {
        node_pair = search(list, value);
        left_node = node_pair->left;
        right_node = node_pair->right;
        if (CAT(left_node->next, right_node, new_node)) {
            free(node_pair);
            return 1;
        }
        set_next(right_node_next, right_node);
        if ((CAT(right_node_next->next, right_node, new_node))) {
            free(node_pair);
            return 1;
        }
    } while (1);
}
113 if ((right_node == speculate(list->tail)) || (right_node->value != value)) {
114     // right_node is a dummy node or doesn't exist in the list
115     return 0; // Deletion failed but element not in list
116 }
117 right_node_next = get_next(right_node);
118 if (!is_marked(right_node_next)) {
119     // Logical deletion
120     if (mark(right_node->next, right_node_next)) {
121         break;
122     }
123 }
124 } while (1);
125 // Physical deletion
126 CAT(left_node->next, right_node, right_node_next);
127 return 1;
128 }
129
130 int find(list *list, int value) {
131     stym_node *right_node;
132     node_pair *node_pair;
133     node_pair = search(list, value);
134     right_node = node_pair->right;
135     if ((right_node == speculate(list->tail)) || (get_value(right_node) != value)) {
136         return 0;
137     } else {
138         return 1;
139     }
140 }
141 }
142
143 node_pair *search(list *list, int value) {
144     node_pair *pair = malloc(sizeof(node_pair));
145     stym_node *left_node_next, *right_node, *left_node;
146     search_again:
147     while (1) {
148         stym_node *t = speculate(list->head);
stym_node *t_next = get_next(speculate(list->head));

// 1: Find left_node and right_node
do {
    if (!is_marked(t_next)) { // If t is not marked
        left_node = t;
        left_node_next = t_next;
    }
    // Move t one node forward
    t = (stym_node*) get_unmarked_reference(t_next);
    // Exit loop if t is the last node
    if (t == speculate(list->tail)) break;
    // Update t_next
    t_next = get_next(t);
} while(is_marked(t_next) || (t->value < value));
// Exit loop if t->value >= value or if t is marked

right_node = t; // right_node = the node being searched for

// 2: Check nodes are adjacent
if (left_node_next == right_node) {
    if (((right_node != speculate(list->tail)) &&
         is_marked(get_next(right_node)))) {
        // Search again if right_node is marked and is not the last (dummy) node.
        goto search_again;
    }
    else {
        pair->left = left_node;
        pair->right = right_node;
        return pair;
    }
}
}
// 3: Physically remove one or more marked nodes
CAT(left_node->next, left_node_next, right_node);
// Before, it was checked whether right_node was unmarked and not the last node.
// If this was the case, left_node and right_node was returned. Now a new loop
// need to proceed since right_node == NULL.
goto search_again;
}
B.4 Tree

B.4.1 Tree C Implementation

```c
#include "tree.h"
#include <limits.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

enum { inf2 = INT_MAX, inf1 = INT_MAX - 1};
enum { CLEAN = 0, MARKED = 1, IFLAG = 2, DFLAG = 3 };
enum { INSERT = 0, DELETE = 1 };

typedef struct info {
    int type;
    node *parent;
    node *leaf;
    node *new_internal;
    node *grand_parent;
    info *pinfo;
} info;

typedef struct node {
    int is_leaf;
    int key;
    info *info;
    node *left, *right;
} node;

typedef struct search_return {
    node *grand_parent;
    node *parent;
    node *leaf;
    info *pinfo;
    info *gpinfo;
} search_return;

node *init() {
    node *root = malloc(sizeof(node));
```
root->info = malloc(sizeof(info));
root->left = malloc(sizeof(node));
root->right = malloc(sizeof(node));
root->left->info = malloc(sizeof(info));
root->right->info = malloc(sizeof(info));

root->key = inf2; // infinity_2
(root->right)->key = inf2;
(root->left)->key = inf1;
(root->right)->is_leaf = 1;
(root->left)->is_leaf = 1;
root->is_leaf = 0;
return root;
}

int get_state(info *pointer) {
  int state = 0;
  int bit0 = ((intptr_t) pointer >> 0) & 1;
  int bit1 = ((intptr_t) pointer >> 1) & 1;
  state |= bit0;
  state |= 2*bit1;
  return state;
}

info *set_state(info *pointer, int mark) {
  int bit0 = ((intptr_t) mark >> 0) & 1;
  int bit1 = ((intptr_t) mark >> 1) & 1;
  intptr_t temp = (intptr_t) pointer;
  temp ^= (-bit0 ^ temp) & (1 << 0);
  temp ^= (-bit1 ^ temp) & (1 << 1);
  //pointer = (int*) temp;
  return (info*) temp;
}

info *get_ptr(info *pointer) {
  intptr_t temp = (intptr_t) pointer;
  temp ^= (0 ^ temp) & (1 << 0);
  temp ^= (0 ^ temp) & (1 << 1);
  return (info*) temp;
}
int max(int x, int y) {
    if (x > y) { return x; } else { return y; }
}

search_return search(int key, node *root) {
    node *grand_parent, *parent;
    node *leaf = root;
    info *gpinfo, *pinfo;
    while (leaf->is_leaf == 0) {
        grand_parent = parent;
        parent = leaf;
        gpinfo = pinfo;
        pinfo = parent->info;
        if (key < leaf->key) {
            leaf = parent->left;
        } else {
            leaf = parent->right;
        }
    }
    search_return ret_struct = {grand_parent, parent, leaf, pinfo, gpinfo};
    return ret_struct;
}

int find(int key, node *root) {
    search_return search_result = search(key, root);
    if ((search_result.leaf)->key == key) {
        return 1; } else {
        return 0;
    }
}

void help_insert(info *op) {
    info *op_ptr = get_ptr(op);
    CAS_child(op_ptr->parent, op_ptr->leaf, op_ptr->
    new_internal); // ichild CAS
    __sync_bool_compare_and_swap(&(op_ptr->parent->info),
    , set_state(op, IFLAG), set_state(op, CLEAN));
    // iunflag CAS
}
int help_delete(info *op) {
    info *op_ptr = get_ptr(op);
    info *result = __sync_val_compare_and_swap(&(op_ptr->parent->info), op_ptr->pinfo, set_state(op, MARKED)); //Mark parent
    if (result == op_ptr->pinfo || result == set_state(op_ptr->parent->info, MARKED)) { //op-parent successfully marked
        help_marked(op); // Complete the deletion
        return 1;
    } else {
        help(result);
        __sync_bool_compare_and_swap(&(op_ptr->grand_parent->info), set_state(op, DFLAG), set_state(op, CLEAN)); //Backtrack
        return 0;
    }
}

void help_marked(info *op) {
    // Precond: op points to a DIinfo record
    node *other;
    info *op_ptr = get_ptr(op);
    if (op_ptr->parent->right == op_ptr->leaf) { other = op_ptr->parent->left; }
    else { other = op_ptr->parent->right; }
    CAS_child(op_ptr->grand_parent, op_ptr->parent, other);
    __sync_bool_compare_and_swap(&(op_ptr->grand_parent->info), set_state(op, DFLAG), set_state(op, CLEAN));
}

// Set other to point to the sibling of the node to which op->leaf points

void help(info *op) {
    int flag = get_state(op);
    if (flag == IFLAG) { help_insert(op); }
    if (flag == MARKED) { help_marked(op); }
    if (flag == DFLAG) { help_delete(op); }
}
int delete(int key, node *root) {
    node *parent, *grand_parent, *leaf;
    info *result, *op, *pinfo, *gpinfo;

    while(1) {
        search_return search_result = search(key, root);
        leaf = search_result.leaf;
        parent = search_result.parent;
        grand_parent = search_result.grand_parent;
        pinfo = search_result.pinfo;
        gpinfo = search_result.gpinfo;

        if (leaf->key != key) { return 0; }
        if (get_state(gpinfo) != CLEAN) { help(gpinfo); }
        else if (get_state(pinfo) != CLEAN) { help(pinfo); }
        else {
            op = malloc(sizeof(info));
            op->parent = parent;
            op->grand_parent = grand_parent;
            op->leaf = leaf;
            op->pinfo = pinfo;
            result = __sync_val_compare_and_swap(&(
                grand_parent->info), gpinfo, set_state(op, DFLAG));
            if (result == gpinfo) {
                if (help_delete(op)) { return 1; }
                else {
                    help(result);
                }
            }
        }
    }
}

int insert(int key, node *root) {
    node *parent, *leaf, *new_sibling, *new_internal;
    info *result, *pinfo;
    node *new = malloc(sizeof(node));
    search_return search_result;
new->key = key;
new->is_leaf = 1;
while(1) {
    search_result = search(key, root);
    leaf = search_result.leaf;
    parent = search_result.parent;
    pinfo = search_result.pinfo;

    if (leaf->key == key) { return 0; }
    int mark = get_state(pinfo);
    if (mark != CLEAN) { help(pinfo); } else {
        new_sibling = malloc(sizeof(node));
        new_sibling->is_leaf = 1;
        new_sibling->key = leaf->key;

        new_internal = malloc(sizeof(node));
        new_internal->is_leaf = 0;
        new_internal->key = max(key, leaf->key);
        (new_internal->info) = malloc(sizeof(info));
        (new_internal->info) = set_state((new_internal->
            info), CLEAN);

        if (new->key < new_sibling->key) {
            new_internal->left = new; new_internal->right =
            new_sibling; } else {
            new_internal->left = new_sibling; new_internal
            ->right = new; }

        info *op = malloc(sizeof(info));
        op->type = INSERT;
        op->parent = parent;
        op->leaf = leaf;
        op->new_internal = new_internal;

        result = __sync_val_compare_and_swap(&(parent->
            info), pinfo, set_state(op, IFLAG));
        if (result == pinfo) {
            help_insert(op);
            return 1; }
else {
    int flag = get_state(parent->info);
    if(flag != 0) {
        help(result); }
    }
}

void CAS_child(node *parent, node *leaf, node* new_internal) {
    //Precond: parent points to an internal node and 
    //new_internal points to a Node.
    //This function tries to change one of the child 
    //fields of the node that parents points to from 
    //leaf to new_internal 
    if (new_internal->key < parent->key) {
        __sync_bool_compare_and_swap(&(parent->left), leaf, new_internal); } else {
        __sync_bool_compare_and_swap(&(parent->right), leaf, new_internal); }
B.4.2 Tree Capable Implementation

```c
#include "tree_kappa.h"
#include <limits.h>
#include <pthread.h>
#include <stdio.h> /* I/O functions: printf() ... */
#include <stdlib.h> /* rand(), srand() */
#include <stdint.h>

#define CAT(a,b,c) (__sync_bool_compare_and_swap(&a,(node*) b, (node*) c) ? c = NULL, 1 : 0)
#define CAT_info(a,b,c) (__sync_bool_compare_and_swap (&a,(info*) b, (info*) c) ? c = NULL, 1 : 0)
#define CAS(a,b,c) (__sync_bool_compare_and_swap(&a,(node*) b, (node*) c))
#define CAS_val(a,b,c) (__sync_val_compare_and_swap(&a,(info*) b, (info*) c))

enum { inf2 = INT_MAX, inf1 = INT_MAX - 1};
enum { CLEAN = 0, MARKED = 1, IFLAG = 2, DFLAG = 3 };
enum { INSERT = 0, DELETE = 1 };

typedef struct info {
  int type;
  node *parent;
  node *leaf;
  node *new_internal; /* Insert */
  node *grand_parent; /* Delete */
  info *pinfo;
} info;

typedef struct stym_info {
  int type; //0 = Insert info, 1 = Delete info
  node *parent;
  node *leaf;
  node *new_internal;
  node *grand_parent;
  info *pinfo;
```
typedef struct node {
    int is_leaf;
    int key;
    info *info;
    node *left, *right;
} node;

typedef struct stym_node {
    int is_leaf;
    int key;
    info *info;
    node *left, *right;
} stym_node;

typedef struct search_return {
    stym_node *grand_parent;
    stym_node *parent;
    stym_node *leaf;
    stym_info *pinfo;
    stym_info *gpinfo;
} search_return;

node *init() {
    node *root = malloc(sizeof(node));
    root->info = malloc(sizeof(info));
    root->left = malloc(sizeof(node));
    root->right = malloc(sizeof(node));
    root->left->info = malloc(sizeof(info));
    root->right->info = malloc(sizeof(info));
    root->key = inf2; // infinity_2
    (root->right)->key = inf2;
    (root->left)->key = inf1;
    (root->right)->is_leaf = 1;
    (root->left)->is_leaf = 1;
    root->is_leaf = 0;
    return root;
int get_type(stym_info *info) {
    return info->type;
}

stym_node *get_parent(stym_info *info) {
    return (stym_node*)info->parent;
}

stym_node *get_leaf(stym_info *info) {
    return (stym_node*)info->leaf;
}

stym_node *get_new_internal(stym_info *info) {
    return (stym_node*)info->new_internal;
}

stym_node *get_grand_parent(stym_info *info) {
    return (stym_node*)info->grand_parent;
}

stym_info *get_pinfo(stym_info *info) {
    return (stym_info*) info->pinfo;
}

stym_info *speculate_info(info *n) {
    return (stym_info*) n;
}

stym_node *speculate(node *n) {
    return (stym_node*) n;
}

stym_node *get_left(stym_node *n) {
    return (stym_node*) n->left;
}

stym_node *get_right(stym_node *n) {
    return (stym_node*) n->right;
}
stym_info *get_info(stym_node *n) {
    return (stym_info*) n->info;
}

int is_leaf(stym_node *n) {
    return n->is_leaf;
}

int get_key(stym_node *n) {
    return n->key;
}

int get_state(stym_info *pointer) {
    int state = 0;
    int bit0 = ((intptr_t) pointer >> 0) & 1;
    int bit1 = ((intptr_t) pointer >> 1) & 1;
    state |= bit0;
    state |= 2*bit1;
    return state;
}

stym_info *set_state(stym_info *pointer, int mark) {
    int bit0 = ((intptr_t) mark >> 0) & 1;
    int bit1 = ((intptr_t) mark >> 1) & 1;
    intptr_t temp = (intptr_t) pointer;
    temp ^= (~bit0 ^ temp) & (1 << 0);
    temp ^= (~bit1 ^ temp) & (1 << 1);
    //pointer = (int*) temp;
    return (stym_info*) temp;
}

stym_info *get_ptr(stym_info *pointer) {
    intptr_t temp = (intptr_t) pointer;
    temp ^= (0 ^ temp) & (1 << 0);
    temp ^= (0 ^ temp) & (1 << 1);
    return (stym_info*) temp;
}

int max(int x, int y) {
    if (x > y) { return x; } else { return y; }
}
search_return search(int key, stym_node *root) {
    stym_node *grand_parent;
    stym_node* parent = NULL;
    stym_node *leaf = root;
    stym_info *gpinfo;
    stym_info* pinfo = NULL;
    while (leaf->is_leaf == 0) {
        grand_parent = parent;
        parent = leaf;
        gpinfo = pinfo;
        pinfo = get_info(parent);
        if (key < leaf->key) {
            leaf = get_left(parent);
        } else {
            leaf = get_right(parent);
        }
    }
    search_return ret_struct = {grand_parent, parent, leaf, pinfo, gpinfo};
    return ret_struct;
}

int find(int key, node *root) {
    search_return search_result = search(key, speculate(root));
    if (((search_result.leaf)->key == key) {
        return 1; } else {
        return 0;
    }
}
stym_info *mark(info *orig, stym_info *ref) {
    stym_info *marked_info = set_state(ref, MARKED);
    return (stym_info*) CAS_val(orig, ref, marked_info);
}
stym_info *clean(info *orig, stym_info *ref) {
    stym_info *marked_info = set_state(ref, CLEAN);
    return (stym_info*) CAS_val(orig, ref, marked_info);
}
stym_info *iflag(stym_info *info, stym_info *orig, stym_info *ref) {
    stym_info *marked_info = set_state(ref, IFLAG);
    return (stym_info*) CAS_val(orig, ref, marked_info);
}

stym_info *dflag(stym_info *info, stym_info *orig, stym_info *ref) {
    stym_info *marked_info = set_state(ref, DFLAG);
    return (stym_info*) CAS_val(orig, ref, marked_info);
}

void help_insert(stym_info *op) {
    stym_info *op_ptr = get_ptr(op);
    CAT_child(op_ptr->parent, get_leaf(op_ptr),
        get_new_internal(op_ptr)); // ichild CAT
    clean(op_ptr->parent->info, op); // iunflag
}

int help_delete(stym_info *op) {
    stym_info *op_ptr = get_ptr(op);
    stym_info *result = mark(op_ptr->parent->info,
        get_pinfo(op_ptr));
    if (result == get_pinfo(op_ptr) || result ==
        set_state(get_info(get_parent(op_ptr)), MARKED))
    {
        help_marked(op); // Complete the deletion
        return 1;
    }
    else {
        help(result);
        // What if this clean fails?
        clean(op_ptr->grand_parent->info, op);
        return 0;
    }
}

void help_marked(stym_info *op) {
    // Precond: op points to a DIIinfo record
    stym_node *other = get_left(get_parent(op_ptr));
}
else {
    other = get_right(get_parent(op_ptr));
}
CAT_child(op_ptr->grand_parent, get_parent(op_ptr),
          other);
clean(op_ptr->grand_parent->info, op);
}

void help(stym_info *op) {
    int flag = get_state(op);
    if (flag == IFLAG) { help_insert(op); }
    if (flag == MARKED) { help_marked(op); }
    if (flag == DFLAG) { help_delete(op); }
}

int delete(int key, node *root) {
    stym_node *parent, *grand_parent, *leaf;
    stym_info *result, *op, *pinfo, *gpinfo;

    while(1) {
        search_result = search(key,
                               speculate(root));
        leaf = search_result.leaf;
        parent = search_result.parent;
        grand_parent = search_result.grand_parent;
        pinfo = search_result.pinfo;
        gpinfo = search_result.gpinfo;

        if (leaf->key != key) { return 0; }
        if (get_state(gpinfo) != CLEAN) { help(gpinfo); }
        else if (get_state(pinfo) != CLEAN) { help(pinfo); }
        else {
            op = malloc(sizeof(info));
            op->parent = (node*) parent;
            op->grand_parent = (node*) grand_parent;
            op->leaf = (node*) leaf;
            op->pinfo = (info*) pinfo;
            result = dflag(grand_parent->info, gpinfo);
            if (result == gpinfo) {
                if (help_delete(op)) { return 1; }
            } else {
        // printf("Delete collision!\n");
}
int insert(int key, node *root) {
    stym_node *parent, *leaf, *new_sibling, *new_internal;
    stym_info *result, *pinfo;
    stym_node *new = malloc(sizeof(node));
    search_return search_result;
    new->key = key;
    new->is_leaf = 1;
    while (1) {
        search_result = search(key, speculate(root));
        leaf = search_result.leaf;
        parent = search_result.parent;
        pinfo = search_result.pinfo;
        if (leaf->key == key) { return 0; }
        int mark = get_state(pinfo);
        if (mark != CLEAN) { help(pinfo); } else {

            new_sibling = malloc(sizeof(node));
            new_sibling->is_leaf = 1;
            new_sibling->key = leaf->key;

            new_internal = malloc(sizeof(node));
            new_internal->is_leaf = 0;
            new_internal->key = max(key, leaf->key);
            new_internal->info = malloc(sizeof(info));

            if (new->key < new_sibling->key) {
                new_internal->left = (node*) new;
                new_internal->right = (node*) new_sibling;
            } else {
                new_internal->left = (node*) new_sibling;
                new_internal->right = (node*) new;
            }
        }
    }
}
info *op = malloc(sizeof(info));
op->type = INSERT;
op->parent = (node*) parent;
op->leaf = (node*) leaf;
op->new_internal = (node*) new_internal;  // New "parent" replacing a leaf
result = iflag(parent->info, pinfo);
if (result == pinfo) {
    help_insert(speculate_info(op));
    return 1;
} else {
    help(result);
}
}

void CAT_child(node *parent, stym_node *leaf, stym_node *new_internal) {
    //Precond: parent points to an internal node and new_internal points to a Node.
    //This function tries to change one of the child fields of the node that parents points to from leaf to new_internal
    if (new_internal->key < parent->key) {
        CAT(parent->left, leaf, new_internal);
    } else {
        CAT(parent->right, leaf, new_internal);
    }
}
B.5 Hash

B.5.1 Hash C Implementation

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include "hash_kappa.h"

#define CAS(a,b,c) __sync_bool_compare_and_swap(&a,b,c)
#define MAX_ALLOWED_BUCKET_SIZE 8

enum { INS = 0, REM = 1, UPSIZE = 2, DOWNSIZE = 3};

typedef struct FSetNode {
    list *set;
    int ok;
} FSetNode;

typedef struct FSet {
    FSetNode *node;
    int element_count;
    int size;
} FSet;

typedef struct FSetOP {
    int type;
    int key;
    int done;
    int response;
} FSetOP;

typedef struct HNode {
    int size;
    int bucket_count;
    int max_bucket_size;
    int op_count;
    struct HNode *pred;
} FSet **buckets;
```
typedef struct HNodeHead {
    HNode *head;
} HNodeHead;

HNodeHead *init_hash() {
    pthread_t Q = pthread_self();
    int start_size = 1;
    HNodeHead *head_root = malloc(sizeof(HNodeHead));
    HNode *head = malloc(sizeof(HNode));
    head_root->head = head;
    head->buckets = malloc(start_size*sizeof(int));
    head->max_bucket_size = 0;
    head->size = start_size;
    head->pred = NULL;
    for(int i = 0; i < start_size; i++) {
        head->buckets[i] = malloc(sizeof(FSet));
        head->buckets[i]->node = malloc(sizeof(FSetNode));
        head->buckets[i]->node->set = init_list();
        head->buckets[i]->node->ok = 1;
    }
    return head_root;
}

int get_response(FSetOP *op) {
    return op->response;
}

int has_member(FSet *bucket, int k) {
    FSetNode *node = bucket->node;
    return find(node->set, k);
}

list *copy_elements(list *new_set, list *old_set) {
    node *n = old_set->head->next;
    while(n != old_set->tail) {
        insert(new_set, n->key);
        n = n->next;
        if(is_marked(n) == 1) {
            n = old_set->head->next;
        } else {
            // continue...
        }
    }
}
```c
list *remove_elements(list *set, int size, int i) {
    node *n = set->head->next;
    while (n != NULL) {
        if (n->key % size != i && n != set->tail) {
            delete(set, n->key);
            n = set->head->next;
        } else {
            n = n->next;
        }
    }
    return set;
}

list *freeze(FSet *b) {
    pthread_t Q = pthread_self();
    FSetNode *o = b->node;
    FSetNode *n;
    while (o->ok) {
        n = malloc(sizeof(FSetNode));
        n->set = o->set;
        n->ok = 0;
        if (CAS(b->node, o, n)) {
            break;
        } else {
            o = b->node;
        }
    }
    return o->set;
}

void update_max_bucket_size(HNodeHead *head_root, FSet *bucket, int result, int type) {
    pthread_t Q = pthread_self();
    HNode *head = head_root->head;
    if (result == 1) {
        if (type == REM) {
            bucket->element_count--;
        }
    }
    while (head->next != NULL) {
        head = head->next;
    }
    if (head->next == NULL) {
        pthread_mutex_lock(&head_root->mutex);
        // Update max_bucket_size
        pthread_mutex_unlock(&head_root->mutex);
    }
}
```
else if (type == INS) {
    bucket->element_count++;
}
if (bucket->element_count > head->max_bucket_size) {
    head->max_bucket_size = bucket->element_count;
}
}
}

int invoke(HNodeHead *head_root, FSet *bucket, FSetOp *op) {
    pthread_t q = pthread_self();
    FSetNode *node = bucket->node;
    FSetNode *new_node;
    int response, result;
    while (node->ok) {
        if (op->type == INS) {
            response = insert(node->set, op->key);
        } else if (op->type == REM) {
            response = delete(node->set, op->key);
        }
        update_max_bucket_size(head_root, bucket, op->type);
        new_node = malloc(sizeof(FSetNode));
        new_node->set = node->set;
        new_node->ok = 1;
        if (CAS(bucket->node, node, new_node)) {
            op->response = response;
            return 1;
        }
        node = bucket->node;
    }
    return 0;
}

FSet *initBucket(HNodeHead *head_root, HNode *current_head, int i) {
    pthread_t q = pthread_self();
    FSet *m, *n, *new_bucket;
    FSetNode *new_node;
```c
list *new_set, *old_set, *old_set_1, *old_set_2;
int CAS_result = 0;
FSet *bucket = current_head->buckets[i];
HNode *pred_head = current_head->pred;
if(bucket == NULL && pred_head != NULL) {
    new_set = init_list();
    if(current_head->size == pred_head->size*2) { // Bigger than pred
        m = pred_head->buckets[i % pred_head->size];
        old_set = freeze(m);
        new_set = copy_elements(new_set, old_set);
        new_set = remove_elements(new_set, current_head->size, i);
    } else { // Smaller than pred
        m = pred_head->buckets[i];
        n = pred_head->buckets[i + current_head->size];
        old_set_1 = freeze(m);
        old_set_2 = freeze(n);
        new_set = copy_elements(new_set, old_set_1);
        new_set = copy_elements(new_set, old_set_2);
    }
    new_bucket = malloc(sizeof(FSet));
    new_node = malloc(sizeof(FSetNode));
    new_bucket->node = new_node;
    new_bucket->node->set = new_set;
    new_bucket->node->ok = 1;
    CAS_result = CAS(current_head->buckets[i], NULL, new_bucket);
}
return current_head->buckets[i];
}

int apply(HNodeHead *head_root, int type, int k) {
    pthread_t Q = pthread_self();
    HNode *head = head_root->head;
    HNode *current_head;
    FSetOP *op = malloc(sizeof(FSetOP));
    FSet *bucket;
    op->type = type;
    op->key = k;
    op->response = 0;
```
op->done = 0;
int i = 0;
while(1) {
    current_head = head_root->head;
    bucket = current_head->buckets[k % current_head->size];
    if(bucket == NULL) {
        bucket = initBucket(head_root, current_head, k % current_head->size);
    }
    if(invoke(head_root, bucket, op)) {
        return get_response(op);
    }
}

int resize(HNodeHead *head_root, int grow) {
pthread_t Q = pthread_self();
HNode *head = head_root->head;
int size = 0;
int success = 0;
FSet **buckets;
HNode *new_head;
HNode *current_head = head;
if (current_head->size > 1 || grow == UPSIZE) {
    for(int i = 0; i < current_head->size; i++) {
        initBucket(head_root, current_head, i);
    }
    current_head->pred = NULL;
    if(grow == UPSIZE) {
        size = current_head->size*2;
    } else if (grow == DOWNSIZE) {
        size = current_head->size/2;
    }
    buckets = malloc(size*sizeof(FSet));
    new_head = malloc(sizeof(HNode));
    new_head->buckets = buckets;
    new_head->op_count = 0;
    new_head->max_bucket_size = 0;
    new_head->pred = current_head;
new_head->size = size;
success = CAS(head_root->head, current_head,
new_head);
}
return success;
}

int insert_hash(HNodeHead *head_root, int k) {
    pthread_t Q = pthread_self();
    HNode *head = head_root->head;
    head->op_count++;
    int response = apply(head_root, INS, k);
    if(head->max_bucket_size > MAX_ALLOWED_BUCKET_SIZE)
    {
        resize(head_root, UPSIZE);
    }
    return response;
}

int delete_hash(HNodeHead *head_root, int k) {
    pthread_t Q = pthread_self();
    HNode *head = head_root->head;
    head->op_count++;
    int response = apply(head_root, REM, k);
    if(1) {
        resize(head_root, DOWNSIZE);
    }
    return response;
}

int contains(HNodeHead *head_root, int k) {
    pthread_t Q = pthread_self();
    HNode *head = head_root->head;
    HNode *current_head = head;
    HNode *pred_head;
    FSet *bucket = current_head->buckets[k %
current_head->size];
    if(bucket == NULL) {
        pred_head = current_head->pred;
        if(pred_head != NULL) {

bucket = pred_head->buckets[k % pred_head->size];
} else {
    bucket = current_head->buckets[k % current_head->size];
}
return has_member(bucket, k);
B.5.2 Hash Capable Implementation

```c
#include <pthread.h>
#include <stdio.h> /* I/O functions: printf() ... */
#include <stdlib.h> /* rand(), srand() */
#include <stdint.h>
#include "hash_kappa.h"
#include "list_hash.c"

#define MAX_ALLOWED_BUCKET_SIZE 1

#define CAT_node(a,b,c) (__sync_bool_compare_and_swap(&a,(FSetNode*) b, (FSetNode*) c) ? c = NULL, 1 : 0)
#define CAT_FSet(a,b,c) (__sync_bool_compare_and_swap(&a,(FSet*) b, (FSet*) c) ? c = NULL, 1 : 0)
#define CAT_HNode(a,b,c) (__sync_bool_compare_and_swap(&a,(HNode*) b, (HNode*) c) ? c = NULL, 1 : 0)
#define CAT_FSetNode(a,b,c) (__sync_bool_compare_and_swap(&a,(FSetNode*) b, (FSetNode*) c) ? c = NULL, 1 : 0)
#define CAS_hash(a,b,c) (__sync_bool_compare_and_swap(&a,(node*) b, (node*) c))
#define CAS_val(a,b,c) (__sync_val_compare_and_swap(&a,(info*) b, (info*) c))
#define CAS(a,b,c) (__sync_bool_compare_and_swap(&a,b,c))

enum { INS = 0, REM = 1, UPSIZE = 2, DOWNSIZE = 3};

typedef struct FSetNode {
    list *set;
    int ok;
} FSetNode;

typedef struct FSet {
    FSetNode *node;
    int element_count;
    int size;
};
```
typedef struct FSetOP {
    int type;
    int key;
    int done;
    int response;
} FSetOP;

typedef struct HNode {
    int size;
    int bucket_count;
    int max_bucket_size;
    int op_count;
    struct HNode *pred;
    FSet **buckets;
} HNode;

typedef struct HNodeHead {
    HNode *head;
} HNodeHead;

typedef struct stym_FSet {
    FSetNode *node;
    int element_count;
    int size;
} stym_FSet;

typedef struct stym_FSetNode {
    list *set;
    int ok;
} stym_FSetNode;

typedef struct stym_FSetOP {
    int type;
    int key;
    int done;
typedef struct stym_HNode {
    int size;
    int bucket_count;
    int max_bucket_size;
    int op_count;
    struct HNode *pred;
    FSet **buckets;
} stym_HNode;

typedef struct stym_HNodeHead {
    HNode *head;
} stym_HNodeHead;

HNodeHead *init_hash() {
    int start_size = 1;
    HNodeHead *head_root = malloc(sizeof(HNodeHead));
    HNode *head = malloc(sizeof(HNode));
    head_root->head = head;
    head->buckets = malloc(start_size*sizeof(int));
    head->max_bucket_size = 0;
    head->size = start_size;
    head->pred = NULL;
    for(int i = 0; i < start_size; i++) {
        head->buckets[i] = malloc(sizeof(FSet));
        head->buckets[i]->node = malloc(sizeof(FSetNode));
        head->buckets[i]->node->set = init_list();
        head->buckets[i]->node->ok = 1;
    }
    return head_root;
}

stym_HNode *get_head(stym_HNodeHead *n) {
    return (stym_HNode*) n->head;
}

stym_list *get_set(stym_FSetNode *n) {
return (stym_list*) n->set;
}

stym_HNodeHead *speculate_HNodeHead(HNodeHead *n) {
    return (stym_HNodeHead*) n;
}

stym_FSetNode *speculate_FSetNode(FSetNode *n) {
    return (stym_FSetNode*) n;
}

stym_FSet *speculate_FSet(FSet *n) {
    return (stym_FSet*) n;
}

stym_FSetOP *speculate_FSetOP(FSetOP *n) {
    return (stym_FSetOP*) n;
}

stym_HNode *speculate_HNode(HNode *n) {
    return (stym_HNode*) n;
}

int get_response(stym_FSetOP *op) {
    return op->response;
}

int has_member(stym_FSet *bucket, int k) {
    FSetNode *node = bucket->node;
    return find((stym_list*) node->set,k);
}

void increment_count(stym_HNode *n) {
    int count = n->op_count;
    CAS(n->op_count, count, count + 1);
}

stym_FSet *get_bucket(stym_HNode *n, int hash) {
return (stym_FSet*) n->buckets[hash];
}

stym_HNode *get_pred(stym_HNode *n) {
    return (stym_HNode*) n->pred;
}

stym_FSetNode *get_node(stym_FSet *n) {
    return (stym_FSetNode*) n->node;
}

int get_size(stym_HNode *n) {
    return n->size;
}

int get_element_count(stym_FSet *n) {
    return n->element_count;
}

list *copy_elements(list *new_set, stym_list *old_set) {
    node *n = old_set->head->next;
    while(n != old_set->tail) {
        insert((stym_list*) new_set, n->key);
        n = n->next;
        if(is_marked(n) == 1) {
            n = old_set->head->next;
        }
    }
    return new_set;
}

list *remove_elements(list *set, int size, int i) {
    node *n = set->head->next;
    while(n != NULL) {
        if(n->key % size != i && n != set->tail) {
            delete((stym_list*) set, n->key);
            n = set->head->next;
        } else {
            n = n->next;
        }
    }
}
stym_list *freeze(stym_FSet *b) {
    stym_FSetNode *o = get_node(b);
    stym_FSetNode *n = NULL;
    while(o->ok) {
        free(n);
        n = malloc(sizeof(FSetNode));
        n->set = o->set;
        n->ok = 0;
        if(CAT_node(b->node, o, n)) {
            break;
        }
        o = get_node(b);
    }
    return get_set(o);
}

void update_max_bucket_size(stym_HNodeHead *head_root, stym_FSet *bucket, int result, int type) {
    stym_HNode *head = get_head(head_root);
    if(result == 1) {
        if(type == REM) {
            bucket->element_count--;
        } else if (type == INS) {
            bucket->element_count++;
        }
        if(bucket->element_count > head->max_bucket_size) {
            head->max_bucket_size = get_element_count(bucket);
        }
    }
}

int invoke(stym_HNodeHead *head_root, stym_FSet *bucket, stym_FSetOP *op) {
    stym_FSetNode *node = get_node(bucket);
    stym_FSetNode *new_node;
int response;
while (node->ok) {
    if (op->type == INS) {
        response = insert(get_set(node), op->key);
    } else if (op->type == REM) {
        response = delete(get_set(node), op->key);
    }
    update_max_bucket_size(head_root, bucket, response, op->type);
    new_node = malloc(sizeof(FSetNode));
    new_node->set = node->set;
    new_node->ok = 1;
    if (CAT_FSetNode(bucket->node, node, new_node)) {
        op->response = response;
        return 1;
    }
    node = get_node(bucket);
} return 0;

stym_FSet *initBucket(stym_HNode *current_head, int i) {
    stym_FSet *m, *n, *new_bucket;
    FSetNode *new_node;
    list *new_set;
    stym_list *old_set, *old_set_1, *old_set_2;
    stym_FSet *bucket = get_bucket(current_head, i);
    stym_HNode *pred_head = get_pred(current_head);
    if (bucket == NULL && pred_head != NULL) {
        new_set = init_list();
        if (current_head->size == pred_head->size * 2) { // Bigger than pred
            m = get_bucket(pred_head, i % pred_head->size);
            old_set = freeze(m);
            new_set = copy_elements(new_set, old_set);
            new_set = remove_elements(new_set, current_head
            ->size, i);
        } else { // Smaller than pred
            m = get_bucket(pred_head, i);
        }
    }
}
n = get_bucket(pred_head, i + current_head->size);
old_set_2 = freeze(n);
old_set_1 = freeze(m);
new_set = copy_elements(new_set, old_set_1);
new_set = copy_elements(new_set, old_set_2);
new_bucket = malloc(sizeof(FSet));
new_node = malloc(sizeof(FSetNode));
new_bucket->node = new_node;
new_bucket->node->set = new_set;
new_bucket->node->ok = 1;
CAT_FSet(current_head->buckets[i], NULL, new_bucket);
return get_bucket(current_head, i);

int apply(stym_HNodeHead *head_root, int type, int k) {
    stym_HNode *current_head;
    FSetOP *op = malloc(sizeof(FSetOP));
    stym_FSet *bucket;
    op->type = type;
    op->key = k;
    op->response = op->done = 0;
    while(1) {
        current_head = get_head(head_root);
        bucket = get_bucket(current_head, k % get_size(current_head));
        if(bucket == NULL) {
            bucket = initBucket(current_head, k % current_head->size);
        }
        if(invoke(head_root,bucket, (stym_FSetOP*) op)) {
            return get_response((stym_FSetOP*) op);
        }
    }
}

int resize(stym_HNodeHead *head_root, int grow) {

int size = 0;
int success = 0;
FSet **buckets;
stym_HNode *new_head;
stym_HNode *current_head = get_head(head_root);
if (current_head->size > 1 || grow == UPSIZE) {
    for(int i = 0; i < current_head->size; i++) {
        initBucket(current_head, i);
    }
    current_head->pred = NULL;
    if(grow == UPSIZE) {
        size = current_head->size*2;
    } else if (grow == DOWNSIZE) {
        size = current_head->size/2;
    }
}

buckets = malloc(size*sizeof(FSet));
new_head = malloc(sizeof(HNode));
new_head->buckets = buckets;
new_head->op_count = 0;
new_head->max_bucket_size = 0;
new_head->pred = (HNode*) current_head;
new_head->size = size;
success = CAT_HNode(head_root->head, current_head, new_head);
}
return success;
}

int insert_hash(HNodeHead *head_root, int k) {
    stym_HNodeHead *stym_head_root = speculate_HNodeHead (head_root);
stym_HNode *head = get_head(stym_head_root);
increment_count(head);
int response = apply(stym_head_root, INS, k);
if(head->max_bucket_size > MAX_ALLOWED_BUCKET_SIZE) {
    resize(stym_head_root, UPSIZE);
}
return response;
}
```c
int delete_hash(HNodeHead *head_root, int k) {
    stym_HNodeHead *stym_head_root = speculate_HNodeHead(head_root);
    stym_HNode *head = get_head(stym_head_root);
    head->op_count++;
    int response = apply(stym_head_root, REM, k);
    if(1) {
        resize(stym_head_root, DOWNSIZE);
    }
    return response;
}

int contains(HNodeHead *head_root, int k) {
    stym_HNodeHead *stym_head_root = speculate_HNodeHead(head_root);
    stym_HNode *current_head = get_head(stym_head_root);
    stym_HNode *pred_head;
    stym_FSet *bucket = get_bucket(current_head, k %
                                   current_head->size);
    if(bucket == NULL) {
        pred_head = get_pred(current_head);
        if(pred_head != NULL) {
            bucket = get_bucket(pred_head, k % get_size(
                                 pred_head));
        } else {
            bucket = get_bucket(current_head, k % get_size(
                                 current_head));
        }
    }
    return has_member(bucket,k);
}
```