CHAPTER 2

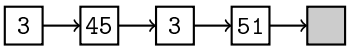
DATA STRUCTURES

2.1 Linked List

A data structure stores data that helps us access them for a particular need. We use a wide variety of data structures since a single data structure isn't useful for all the requirements, e.g. array, set, queue, tree, stack.
Just sticking with arrays is a bad idea. For example, we cannot dynamically add or delete to an array- we would need to resize the array and copy all the elements. Also, a lot of memory might be allocated, but not much of that might be used in an array. For that reason, we seek other dynamic data structures, like a linked list.
A linked list is the simplest type of a dynamic data structure. A linked list is made up of nodes with a pointer to another node(s).

Singly Linked List

A singly linked list, or just a linked list, is a sequence of nodes arranged in linear order. Every node has a pointer to the next node. For example, the following is a linked list:



Every node is represented by a square. The arrow represents whether the attribute `next` of the node points. The tail in the array points to `null`. We refer to the first element in the linked list by `head`. In this case, the node corresponding to `head` has value 3.

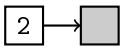
Insert

Now, we consider addition of a node into a linked list at the head. The following is the pseudocode:

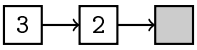
```
void insert<E>(LinkedArray<E> array, Node<E> node):
    node.next = array.head
    array.head = node
```

Here, we set the `next` attribute of the `node` to the `head` of the array. Then, we set the `head` of the array to be the new node. This function has complexity $O(1)$.
Assume we start with an empty linked list and add two nodes with values 2 and 3.

We start with an empty linked list.



To the empty array, we first add a node with value 2. That is, the `next` attribute of the `node` becomes the previous `head` `null`. This also implies that 2 is where the array ends. Then, the `head` of the array becomes the `next`.



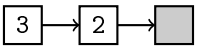
Next, we add the element 3 at the start. 2 gets pushed to the next element, and the array now has 2 elements.

Delete

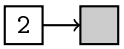
We now consider deletion of a node from the head. The following is the pseudocode:

```
void delete<E>(LinkedArray<E> array):
    if (array.head != null):
        array.head = array.head.next
```

We move the `head` of the array to the `next` element. That way, the `head` element becomes removed from the array. This function has complexity $O(1)$. Assume we now call the `delete` function on the array `[3, 2]` twice.



We start with the array with all 3 elements.



When we first call delete, we set the `head` of the array to be the `next` element 2.

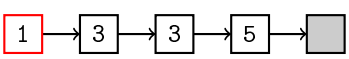
Calling delete again makes the array empty. Any further call to delete doesn't make a change to the array.

Search

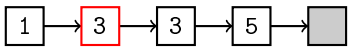
We now consider searching a value within the array. The function returns the first node containing the required value, if it is found. If not, it returns `null`. This is a simple linear search that iterates through the array, finding a node with that value. The pseudocode is:

```
Node<E> search<E>(LinkedArray<E> array, E value):
    Node<E> node = array.head
    while (node != null && node.value != k):
        node = node.next
    return node
```

We show how the function works by finding the node with value 3 from the linked list `[1, 3, 3, 5]`.



Originally, the value of `node` is the first node. Since it isn't a `null` node and its `value` isn't 1, we continue the while loop.



Now, since 3 is the value of this node, we return the node.
If we wanted to find 7 in the array, we would have ended up returning the final, `null` node.
The structure of a linked list allows for us to make it a recursive function. The psuedocode in that case is:

```
void search<E>(Node<E> node, E value):
    if (node == null):
        return node
    else if (node.value == k):
        return node
    return search(node.next, value)
```

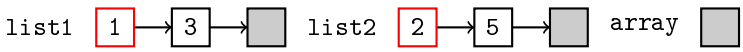
We start the recursive call first with `array.head`.

Search

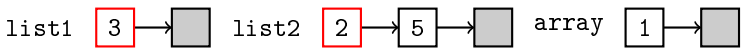
A sorting algorithm for a linked list cannot rely on accessing any index in constant time. So, we use merge sort to sort the linked list. This is a divide and conquer algorithm, involving splitting the array into sorted sublists and then merging the array. We first consider merging two sorted linked lists:

```
Node<int> merge(Node<int> node1, Node<int> node2):
    if (node1 == null):
        return node2
    else if (node2 == null):
        return node1
    Node<E> merged = null
    if (node1.value <= node2.value):
        merged = node1
        merged.next = merge(node1.next, node2)
    else:
        merged = node2
        merged.next = merge(node1, node2.next)
    return merged
```

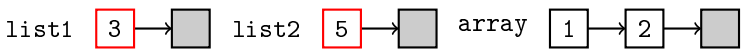
Here, the two nodes are the heads of the two linked lists. The returned node is the head of the merged linked list. It recursively establishes the next element of the node by considering the highest element of the largest elements.
We now consider how the function merges the two arrays `[1, 3]` and `[2, 5]`.



We start with `list1` and `list2` with all the elements, and the merged `array` empty.

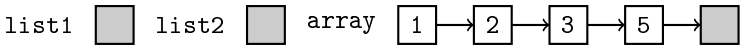


Out of the `head` nodes 1 and 2, 1 was smaller. So, we first add 1 to the `array`.



Since 2 is smaller than 3, we add 2 into the `array`.

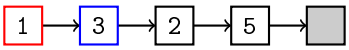
Now, `3` is smaller than `5`, so we add that node.



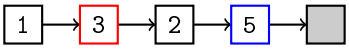
Since `list1` is empty, we add all the elements from `list2` into the `array`. The `array` is now sorted.
We now consider how we split the linked list into halves. We do this using two cursors- one cursor travels one node per iteration, while the second travels two nodes. So, at the end of the loop, the slow cursor is just before the midpoint.

```
[Node<int>, Node<int>] split(Node<int> node):
    if (node == null || a.next == null):
        return [node, null]
    Node<int> slow = node
    Node<int> fast = node.next
    while (fast != null && fast.next != null):
        slow = slow.next
        fast = fast.next.next
    Node<int> mid = slow.next
    slow.next = null
    return [node, mid]
```

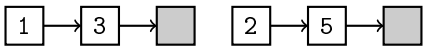
We consider how the function splits the array `[1, 3, 3, 5]`.



This array has at least 2 elements, so we need to search the middle element. At the start, we have the two pointers- `slow` (red box) and `fast` (blue box). Since the blue node isn't the tail of the array, we move both the nodes.



Now, however, the blue node is the final node. So, we break the array at the location of the red node, and get the two arrays.



We return the `head` node of the two arrays.
Finally, we consider how the merge sort makes use of these functions:

```
Node<int> mergeSort(Node<int> node):
    if (node == null || node.next == null):
        return node
    [left, right] = split(node)
    Node<int> sortedl = mergeSort(left)
    Node<int> sortedr = mergeSort(right)
    return merge(sortedl, sortedr)
```

The function is called with the `head` of the linked list, and returns the head of the sorted linked list's `head`.

Tail insertion

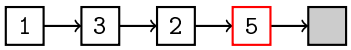
If we wanted to add an element to the tail of the array, we need to traverse through the entire array. Therefore, it has complexity $O(n)$. The psuedocode for the insertion is:

```
void tailInsert<E>(LinkedArray<E> array, Node<E> node):
    if (array.head == null):
        insert(array, node)
    Node<E> prev = array.head
    while (prev.next != null):
        prev = prev.next
    prev.next = node
    node.next = null
```

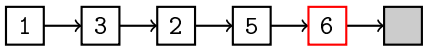
Since this could be a common call on the linked list, it might not be the most efficient algorithm. We could add a pointer to the tail so the algorithm is linear. In that case, we need to update the functions `insert` and `delete`. Then, the `tailInsert` function becomes:

```
void tailInsert<E>(LinkedArray<E> array, Node<E> node):
    node.next = null
    if (array.tail == null):
        array.head = node
    else:
        array.tail.next = node
    array.tail = node
```

We consider how this function works by adding a node containing `6` to the tail of the linked list `[1, 3, 2, 5]`. Originally, we have the array



The node `6` gets added to the tail- the array isn't empty, so we make the `next` of the tail `5` be the node, and then the tail of the array `6`:



Doubly Linked List

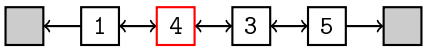
If we want to remove a node from a linked list, then need to find the node before the removing node. The only way we can do this is by traversing the array- the function will have complexity $O(n)$. This would not be an efficient algorithm.
To make deleting an element from a linked list a constant algorithm, we introduce doubly linked lists. Here, every node has not just a `next` attribute, but also `prev`, which points to the previous node in the array. Although this makes many operations more efficient, it means we have to store more in memory.

Deletion

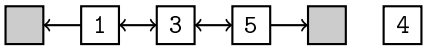
We now consider how to delete a node from a doubly linked list:

```
void delete<E>(Array<E> array, Node<E> node):
    if (node.prev != null):
        node.prev.next = node.next
    else:
        array.head = x.next
    if (node.next != null):
        node.next.prev = node.prev
```

We now visualise how the function works by deleting the node with value `4` from the doubly linked list `[1, 4, 3, 5]`:

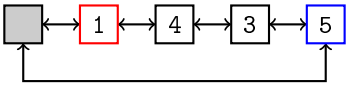


Since `4` isn't the `head`, we change the `next` of `1` to `3`, and the `prev` of `3` to `1`. So, the array becomes:



Circular Linked List

Having boundary conditions complicate the operations on a (doubly) linked list. So, we connect the `head` and the `tail`. For example, the following is a circular doubly linked list:



The first element is called the sentinel `gap`, and has no value- the element before the sentinel is the `tail` (boxed in blue), while the element after the sentinel is the `head` (boxed in red). Using the circular doubly linked with a sentinel, the functions we covered have much fewer conditions. The three functions `delete`, `insert` and `search` are listed below:

```
// delete the specified node from the array
void delete<E>(CLinkedArray<E> array, Node<E> node):
    node.prev.next = node.next
    node.next.prev = node.prev
```

```
// add to the head of the array
void insert<E>(CLinkedArray<E> array, Node<E> node):
    node.next = array.head
    array.head.prev = node
    array.head = node
    node.prev = gap
```

```
// find the node in the array with that value
Node<E> search<E>(CLinkedArray<E> array, E value):
    Node<E> node = array.head
    // search until we find the element or we end up at the gap
    while (node != gap && node.value != value):
        node = node.next
    return node
```

2.2 Abstract Data Types

An abstract data type (ADT) defines a data type and operations on the data type. For example, for the data type array, we expect indexing every element. An ADT is a class of objects whose logical behaviour is defined by a set of values and operations- it is an abstract expectation of the implementation. We make use of data structures to make a concrete implementation of the data type. The ADT is the logic, while the data structure makes it physical.

Stack

The stack ADT stores elements. It allows us to add and delete elements using the last-in-first-out (LIFO) policy. The main operations defined on a stack are:

- `void push<E>(Stack<E> stack, E element)`, which adds the `element` into the stack;
- `E pop<E>(Stack<E> stack)`, which removes and returns the most recently pushed element from the stack.

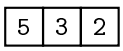
Along with these operations, we also have the following additional operations:

- `E peek<E>(Stack<E> stack)`, which returns the most recently pushed element from the stack without removing it;
- `int size<E>(Stack<E> stack)`, which returns the number of elements in the stack;
- `boolean isEmpty<E>(Stack<E> stack)`, which returns `true` if the stack is empty, and `false` otherwise.

Now, we see how we can perform the operations on the stack. We first push the element `2` into the stack. Then, the stack becomes:

Then, we push `3` onto the stack. The stack is now

We represent pushing an element as adding to the left. It can be represented in any of the other 3 possibilities as well. Now, we push `5` to the stack. Now, the stack is



If we now pop the stack, we get the value `5` out of the stack. Also, the stack is now

Now, we peek the stack- we get the value `3`, but the stack remains unchanged. Next, we pop again. We return `3` by the operation, making the stack

Finally, we push 7 into the stack. The final state of the stack is

Array implementation

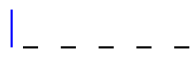
Now, we consider implementations of the stack using an array. Since an array can only be of a particular size, the size of the stack is also limited. In fact, if the array is of length `n`, then the size of the stack is `n`. We keep track of the index `top` to keep track of the top value of the stack.
We cannot pop from an empty stack. This is true for all stacks. On the other hand, using an array implementation, we also cannot push to a full stack. This is not part of the logic of a stack.
Keeping track of the `top` element within the array, the following is the pseudocode for the stack:

```
boolean isEmpty<E>(Stack<E> stack):
    return stack.top == -1
```

```
void push<E>(Stack<E> stack, E value):
    stack.top++
    stack.array[stack.top] = value
```

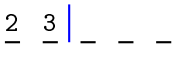
```
E pop(Stack<E> stack):
    if (stack.isEmpty):
        throw UnderflowError
    stack.top--
    return stack.array[stack.top+1]
```

We show how the concrete implementation handles the stack operations we discussed above. We start with an array of size 5.

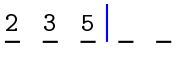


To the left of the blue wall, we have the element `top`- it is initialised with value `-1`. We then push `2` onto the stack:

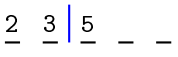
Adding an element increments the value of `top`. Next, we push `3` onto the stack:



We push `5` onto the stack now:



Now, we wish to pop the stack. So, we return `5`, and the stack becomes



Although 5 is still in the array, it doesn't matter since top has been decremented- it gets overridden when we push again. Next, we peek the stack- this returns 3, and the stack remains unchanged. Then, we pop the stack. This returns 3 as well, but the stack changes this time. It is now

$$\underline{2} \mid \underline{3} \;\; \underline{5} \;\; \underline{} \;\; \underline{}$$

Finally, we push 7 onto the stack

$$\underline{2} \;\; \underline{7} \mid \underline{5} \;\; \underline{} \;\; \underline{}$$

So, 3 gets overridden with 7.
All the operations in the array run in constant time. The space used is always $O(n)$, where n is the size of the array. So, n can be much larger than the size of the stack. Also, n must be known at the start.
To overcome these issues, we make use of a resizable array. Here, if we have an overflow, we increase the size of the array. Similarly, if most of the array isn't being used, we decrease the size of the array. We now avoid overflow errors. Expanding the array takes $O(n)$ time overall.
We now consider the pseudocode for resize.

```
void resize<E>(Stack<E> stack, int newLength):
    // newArray has size newLength
    Array<E> newArray = []
    for (int i = 0; i < stack.top; i++):
        newArray[i] = stack.array[i]
    stack.array = newArray
```

Here, we copy all the values from the array in the stack onto the newArray, and then make that array the array of the stack. We now look at how this function is called from the implementations of pop and push.

```
E pop<E>(Stack<E> stack):
    if (stack.isEmpty):
        throw UnderflowError
    E value = stack.array[stack.top]
    stack.top--
    // half the array if only a quarter of it is being used
    if (stack.top > 0 && stack.top == n/4):
        resize(stack, n/2)
```

```
void push<E>(Stack<E> stack, E value):
    // if not enough space, double the array
    if (stack.top == n-1):
        resize(stack, 2*n)
    stack.top++
    stack.array[stack.top] = value
```

For example, if we had the stack

$$\underline{3} \;\; \underline{4} \mid$$

and we wanted to push 5, the stack then becomes

$$\underline{3} \;\; \underline{4} \;\; \underline{5} \mid \underline{}$$

If we now pop twice, then we end up with

When consider the running time of a resizable array implementation, we perform amortised analysis. That is, we consider the average case. For instance, if we want to add `n+1` push operations to the stack, then the first `n` operations are $O(1)$, while the last one is $O(n)$ since we need to resize. On average, this means that the running time is $O(1)$. Therefore, the amortised running time of the push operation is $O(1)$.

Linked list implementation

Now, we consider the linked list implementation. A linked list follows LIFO policy when adding and removing, so it is the same implementation. In particular, the `top` of a stack is the `head` of a linked list. Moreover, all the operations are constant- we can keep track of the `size` attribute in order to make it run in constant time.

Queue

Like a stack, the queue ADT also stores arbitrary elements. However, the insertions and deletions obey first-in-first-out (FIFO) policy. The main operations on a queue are:

- `void enqueue<E>(Queue<E> queue, E value)`, which inserts `value` at the end of the `queue`;
- `E dequeue<E>(Queue<E> queue)`, which removes and returns the element at the head of the `queue`.

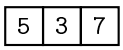
Other operations defined on a queue are:

- `E front<E>(Queue<E> queue)`, which returns the element at the head of the `queue` without removing it;
- `int size<E>(Queue<E> queue)`, which returns the number of elements in the `queue`;
- `boolean isEmpty<E>(Queue<E> queue)`, which returns `true` if there are no elements in the `queue` and `false` if there are elements in the `queue`.

We describe how we these operations apply to the queue. First, to an empty queue, we add `5`.

Next, we add `3` to the queue:

We set the leftmost element of the queue the front, while the rightmost element of the queue the back. This is why `3` gets place to the right of `5`. Now, we add `7` to the queue.



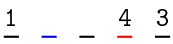
Now, we dequeue the queue. So, we return `5` and the queue becomes

We dequeue again, returning `3`. The queue is now

If we peek at the `front` of the queue, we return `7`, but the queue remains unchanged. If we dequeue, then `7` gets returned, and the queue is empty. Trying to dequeue again raises an error. If we check `isEmpty`, we will get back `true`.

Array implementation

Now, we look at the array implementation of a queue. Due to the nature of an array, the queue has a capacity. The array implementation makes use of two attributes- `head` and `tail` that track the indices of the head and the tail of the queue respectively. The value of `tail` is one further than the actual index of `tail`- it points to the location where the next element will be added. For example, the following is a possible queue:



The blue spot is the location of `tail`, while the red spot marks the `head` of the queue. So, the `tail` is always empty- an array of length `n` implements a queue with maximum length `n-1`. Moreover, the queue is empty if and only if the `head` and the `tail` indices match.
The operations on the array implementation increment the values of `head` and `tail` in a circular manner (i.e. modular addition). We now consider the operations:

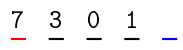
```
boolean isEmpty<E>(Queue<E> queue):
    return queue.head == queue.tail
```

```
int size<E>(Queue<E> queue):
    return (queue.array.length - queue.head + queue.tail) % queue.
    array.length
```

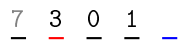
```
void enqueue<E>(Queue<E> queue, E value):
    if (queue.size == n-1):
        throw OverflowError
    queue.array[queue.tail] = value
    queue.tail = (queue.tail + 1) % queue.array.length
```

```
E dequeue<E>(Queue<E> queue):
    if (queue.isEmpty):
        throw UnderflowError
    E value = queue.array[queue.head]
    queue.head = (queue.head + 1) & queue.array.length
    return value
```

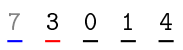
We consider these operations using the array



The queue is full. We dequeue, which returns `7`, and increments the value of `head` modulo `5`.



Note that the value `7` is still in the array. This doesn't matter since the `head` points to the next element `3`. Now, we enqueue `4`. This increments the value of `tail` modulo `5`.



Since `4+1=5`, the value of `tail` becomes `0`. We now compute the `size` of the array, which is `5-1+0=4` modulo `5`, so `4`. Next, we dequeue.

We dequeue again.

Finally, we enqueue `5`.

The space used is always $O(n)$, where n is the length of the array. So, we might be wasting space. Nonetheless, while the operations are $O(1)$. To counter the waste of space, we can make use of a resizable size implementation, the same way we used in stack. Then, all the operations take amortised constant time. The resize function is shown below:

```
void resize<E>(Stack<E> stack, int newLength):
    // array has length newLength
    Array<E> array = []
    for (int i = 0; i < n-1; i++):
        array[i] = queue.array[(queue.head+i) % queue.array.length
    ]
    queue.tail = queue.array.length
    queue.array = array
    queue.head = 0
```

When we resize, we place the elements in the array from `0` to `n-2`. We make use of the resize operation when performing the operations enqueue and dequeue, either halfing or doubling the length of the array.

Linked list implementation

Now, we consider the linked list implementation. The linked list we use here has a tail pointer for efficiency. Then, the `head` and the `tail` of the list and the stack match. Moreover, `enqueue` is implemented by adding to the tail, while `dequeue` is implemented by removing from the head. All the operations are performed in constant time, except for computing the `size`. But, keeping track of the `size` will make it a constant operation. Moreover, there are no overflows since the linked list implementation is dynamic.
Using doubly linked list has the same performance, but is less memory efficient.

Dequeue

The doubly ended queue (dequeue) ADT supports insertion and deletion from both ends. The operations here are:

- `void pushBack<E>(Dequeue<E> dequeue, E value)`, which inserts the `value` at the end of the dequeue;
- `void pushFront<E>(Dequeue<E> dequeue, E value)`, which inserts the `value` at the start of the dequeue;
- `E popBack<E>(Dequeue<E> dequeue)`, which pops the element from the end of the queue.
- `E popFront<E>(Dequeue<E> dequeue)`, which pops the element from the start of the queue.

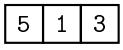
Other operations defined on a dequeue are:

- `E front<E>(Dequeue<E> dequeue)`, which returns the element at the head of the queue without removing it;
- `E back<E>(Dequeue<E> dequeue)`, which returns the element at the back of the queue without removing it;
- `int size<E>(Dequeue<E> dequeue)`, which returns the number of elements in the `dequeue`;
- `boolean isEmpty<E>(Dequeue<E> dequeue)`, which returns `true` if there are no elements in the `dequeue` and `false` if there are elements in the `dequeue`.

We describe how we these operations apply to the dequeue. First, to an empty dequeue, we push `1` to the front.

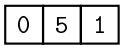
Now, we push `3` to the back of the dequeue.

Next, we push `5` to the front of the dequeue.



If we calculate the `size` of the dequeue, we get `3`. Then, we pop back the dequeue. This returns `3`, and the dequeue becomes

We now push `0` to the front of the dequeue.



The `front` of the dequeue is `0`, while the `back` of the deque is `1`. Finally, we pop front the dequeue. This return `0`, and the dequeue becomes

Array implementation

We now consider the array implementation of a dequeue. This is very similar to the array implementation of queue, just with a few further functions. One of the most important differences is that when we add to an empty dequeue or remove to get an empty dequeue, pushing and popping from the front is the same as doing the operations from the back. The code is shown below:

```
void pushBack<E>(Dequeue<E> dequeue, E value):
    if (dequeue.size == n-1):
        throw OverflowError
    dequeue.array[dequeue.tail] = value
    dequeue.tail = (dequeue.tail+1) % dequeue.array.length
```

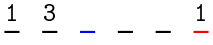
```
void pushFront<E>(Dequeue<E> dequeue, E value):
    if (dequeue.size == n-1):
        throw OverflowError
    // if list empty, add to the tail
    if (dequeue.head == dequeue.tail):
        dequeue.array[dequeue.tail] = value
        dequeue.tail = (dequeue.tail+1) % dequeue.array.length
    // otherwise add to the front
    else:
        dequeue.head = (dequeue.head-1) % dequeue.array.length
        dequeue.array[dequeue.head] = value
```

```
E popBack<E>(Dequeue<E> dequeue):
    if (dequeue.isEmpty):
        throw UnderflowError
    dequeue.tail = (dequeue.tail-1) % dequeue.array.length
    return dequeue.array[dequeue.tail]
```

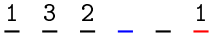
```
E popFront<E>(Dequeue<E> dequeue):
    if (dequeue.isEmpty):
        throw UnderflowError
    // if this is the only element, then pop back not pop front
    if (head+1 == tail):
        dequeue.tail = (dequeue.tail-1) % dequeue.array.length
        return dequeue.array[dequeue.tail]
    else:
        E value = dequeue.array[dequeue.head]
        dequeue.head = (dequeue.head+1) % dequeue.array.length
        return value
```

```
int size<E>(Dequeue<E> dequeue):
    return (dequeue.array.length - dequeue.head + dequeue.tail) %
    dequeue.array.length
```

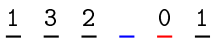
All the operations not shown are the same as a queue. We consider these operations using the array



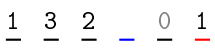
The head here is 1, while the tail is 3- we traverse rightwards in a circular fashion. This is similar to the array implemented by queue. First, we push 2 to the tail of the dequeue.



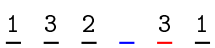
Pushing to the end implies incrementing the `tail` in a circular fashion. Moreover, we add the element to the right, to where the `tail` was. Next, we push `0` to the front.



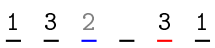
Pushing to the front implies decrementing the `head` in a circular fashion. Moreover, we add the element to the left, the index after the `head`. Now, we pop from the front of the dequeue. This returns `0`, and the array becomes



Like with the queue, the element isn't removed- we just increment `head` in a circular fashion. The `front` of the dequeue is `1`, and the `tail` is `2`. Then, we push `3` to the front.



This decrements the `head` in a circular fashion, and replaces the `0` with `3`. Finally, we pop back from the dequeue. This returns `2`, and the dequeue becomes



Now, the `head` and the `tail` of the dequeue is `3`. The `size` is now `4`.
To overcome the size limitation of the dequeue, we can make use of a resizable array implementation, like in the case of a queue. All the operations, in both cases, takes (amortised) constant running time.

Doubly linked list implementation

We can implement a dequeue using a doubly linked list with a `tail` pointer. Then, popping and pushing an element from the linked list becomes a constant operation. We could even keep track of the `size` when popping and pushing in order to make it a constant operation. Moreover, if we use a circular linked list, we have fewer conditions to check.

List

The list ADT stores a countable sequence of arbitrary elements. Duplicates are allowed within the list. It is one of the fundamental data type in most functional programming languages. The following are the main operations on a list:

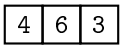
- `E get<E>(List<E> list, int i)`, which returns the element in the `list` at index `i` without removing it;
- `void set<E>(List<E> list, int i, E value)`, which replace the value of the `list` at index `i` with the provided `value`;
- `void add<E>(List<E> list, E value)`, which adds the `value` at the end of the `list`;

- `void insert<E>(List<E> list, int i, E value)`, which adds `value` at index `i` in the `list` and shifts all the elements after the index to the right;
- `E remove<E>(List<E> list, int i)`, which returns and removes the element at index `i` and shifts all the elements after the index to the left.

We describe how we these operations apply to the list. First, to an empty dequeue, we add `4` to the list.

Next, we add `6` to the list

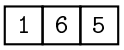
Adding an element pushes to the right. Moreover, the element at index `0` of the list is `4`, while at index `1`, the element is `6`. Now, we add `3` to the list



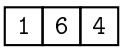
Now, we remove the element from the list at index `0`. This returns `4` and the list is now

We shift the indices of all the elements to the left by `1`. Next, we make the element at the first index `5`.

Now, we add `0` at index `1`. So, the list is now



We shift the indices of all the elements to the right. Finally, at index `2`, we set the value as `4`. The list is now



We also define a further operation to two lists- `concatenate`, with signature `List<E> concatenate<E>(List<E> list1, List<E> list2)`. This returns a list of all the elements in both the lists, starting from the elements in `list1` and then `list2`. For example, if `list1` is `[1, 2, 3]` and `list2` is `[5, 3]`, then the `concatenate` function returns `[1, 2, 3, 5, 3]`.

Resizable array implementation

We now consider the resizable array implementation of the list data structure. They make use of the attribute `size` to keep track of the length of the list. The implementations of get and set directly translate to the array implementation.

```
E get<E>(List<E> list, int i):
    return list.array[i]
```

```
void set<E>(List<E> list, int i, E value):
    list.array[i] = value
```

When adding, we might need to resize the array.

```
void add<E>(List<E> list, E value):
    if (list.size == list.array.length):
        resize(list, 2*list.size)
    list.array[list.size] = value
    list.size++
```

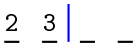
The `resize` function just copies the elements from this array to the other array at the same index. The size of the new array is just different to the previous array.
The `insert` and `delete` functions shift elements to the left or the right. We need to also increment or decrement `end` to keep track of the size of the list.

```
void insert<E>(List<E> list, int i, E value):
    if (list.size == list.array.length):
        resize(list, 2*list.size)
    list.size++
    for (int j = size; j >= i; j--):
        list.array[j+1] = list.array[j]
    list.array[i] = value
```

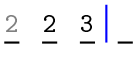
```
E delete<E>(List<E> list, int i):
    E value = list.array[i]
    for (int j = i+1; j < size; j++):
        list.array[j-1] = list.array[j]
    list.size--
    if (list.size < list.array/4):
        resize(list, list.size/2)
    return value
```

If we start with the list

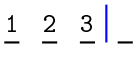
and insert `1` at index `0`, we first resize



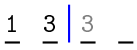
and increment the size and shift the numbers.



Then, we add `1` at index `0`.



If we now delete the element at index 1, then we return 2 and shift the elements after index 1 to the left.



The number 3 remains in the list, but the `size` of the list is 2.
The functions `get`, `set` and `add` have (amortised) running time $O(1)$, while the other functions have running time $O(n)$.
Finally, we consider the concatenation function. It creates another array and adds the elements from the first list before adding those from the other list. The pseudocode is:

```
List<E> concatenate<E>(List<E> list1, List<E> list2):
    // array of length list1.array.length + list2.array.length
    Array<E> array = []
    for (int i = 0; i < list1.size; i++):
        array[i] = list1.array[i]
    for (int i = 0; i < list2.size; i++):
        array[i+list1.size] = list2.array[i]
    return List(array = array, size = list1.size + list2.size)
```

This function has running time $O(n_1 + n_2)$, where n_1 and n_2 are the lengths of the arrays within the two lists `list1` and `list2`.

Linked list implementation

We now consider the implementation of the list using a doubly linked list with a tail pointer. Since we have pointers for `tail`, the function on adding to a linked list is:

```
void add<E>(List<E> list, E value):
    DNode<E> node = DNode(value)
    list.tail.next = node
    node.prev = list.tail
    list.tail = node
```

It runs on constant time. The functions `get` and `set` need to traverse the linked list. Their pseudocode is given below:

```
E get<E>(List<E> list, int i):
    DNode<E> node = list.head
    for (int j = 0; j < i; j++):
        node = node.next
    return node.value
```

```
void set<E>(List<E> list, int i, E value):
    DNode<E> node = list.head
    for (int j = 0; j < i; j++):
        node = node.next
    node.value = value
```

The two functions are linear. Now, we look at `insert` and `delete`.

```
void insert<E>(List<E> list, int i, E value):
    DNode<E> prev = list.head
    DNode<E> node = DNode(value)
    for (int j = 0; j < i-1; j++):
        prev = prev.next
    prev.next.prev = node
    node.next = prev.next
    prev.next = node
    node.prev = prev
```

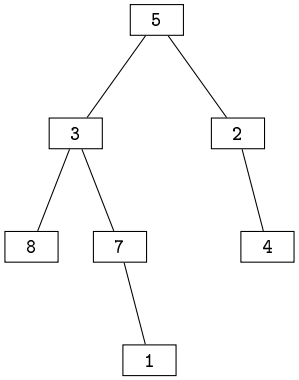
```
E delete<E>(List<E> list, int i):
    DNode<E> node = list.head
    for (int j = 0; j < i; j++):
        node = node.next
    node.prev.next = node.next
    node.next.prev = node.prev
    return node
```

We still need to traverse the list, so the two functions run on linear time. Finally, we consider concatenation of two lists. Since we have a pointer to the `tail` of the linked list, this is a trivial operation- we connect the tail of the first list with the tail of the second list. The psuedocode is

```
List<E> concatenate<E>(List<E> list1, List<E> list2):
    list1.tail.next = list2.head
    list2.head.prev = list1.tail
    return list1
```

2.3 Binary Trees

A binary tree is a linked data structure with nodes. Every node has a key, along with pointers to the `parent`, `left` and `right`. For example, in the following binary tree

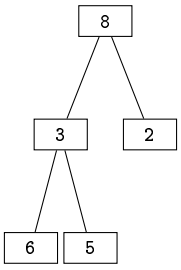


Here, the left child of `3` is `8`, while the right child is `7`. Moreover, its parent is `5`. If a node is missing a pointer, then we set it as `null`. The root of the tree has no parent, i.e. in the tree above, the root is 5. We call a node without any left or right children a leaf.

We define the path between a node and its ancestor as the nodes going from the ancestor to the node. The height of a tree is the length of the longest path from the root. For example, the path from `5` to `4` has length 2. The longest path goes from `5` to `6`, and has height 4.

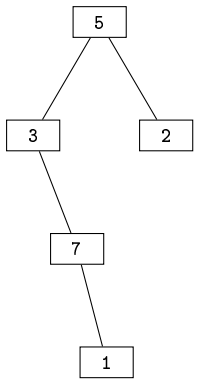
We call a binary tree without duplicates injective. A balanced binary tree is a binary tree where the left and the right subtrees of every node differ in height by no more than 1. In that case, traversal of the height is $O(\log n)$- this allows us to exploit the binary tree structure in different functions. If a tree is extremely unbalanced, it looks like a linked list. So, traversing the height is $O(n)$.

For example, consider the following tree:



The height of the leaves is 0; the difference in the heights of the left and the right subtrees of `3` is 0, while the difference in the heights of the left and the right subtrees of `8` is 1 (`2` has height 0, while `3` has height 1). So, the maximum difference is 1, meaning that the tree is balanced.

Now, consider the following tree:



Here, the height of the left subtree of the root is 2, while the height of the right subtree is 0. The difference is 2, so the tree isn't balanced.

Now, we consider the number of binary trees with n nodes. If $n = 0$, then we just have the empty tree. If $n = 1$, then we have the trivial tree. If $n = 2$, then we can have two possible trees- either a root with a left child, or a root with a right child. If $n = 3$, then there are 5 possibilities- the 2 completely unbalanced trees, the one balanced tree and the two partially-balanced trees (root-left-right, or root-right-left). In general, for a number n, there are C_n number of trees, where C_n refers to the n-th Catalan number. We can recursively define the n-th Catalan number as

$$C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}.$$

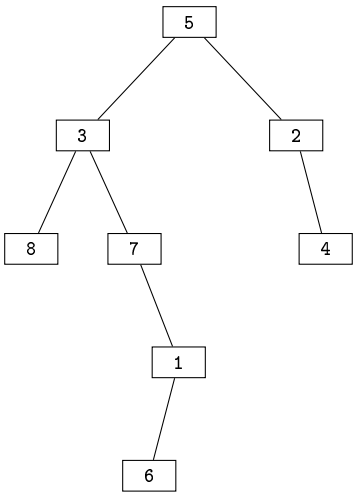
Traversing a binary tree

We now look at 3 ways in which we can traverse the binary tree. When we traverse the tree, we want to visit a node precisely once. There are 3 traversals- inorder, preorder and postorder traversals.

The inorder traversal of a binary tree is given by the following pseudocode

```
void inorder<E>(BNode<E> node, void Function(E) fn):
    if (node != null):
        inorder(node.left)
        fn(node.value)
        inorder(node.right)
```

We initiate the function with the `root`. For example, if the function is to `print` the `value`, then for the binary tree



we get `8, 3, 7, 6, 1, 5, 2, 4`. The running time of this function is $\Theta(n)$, assuming `fn` is a constant function. Since all the nodes are visited, $T(n) = \Omega(n)$. Using the iterative method, we know that $T(0) = O(1)$, and $T(n) = T(k) + T(n-k+1) + O(1)$, where k is the number of nodes in the left subtree (meaning that the right has $n-k-1$ nodes). We can set $k = 0$ to hit the base case on the left subtree, in which case the recurrence equation simplifies to:

$$\begin{aligned} T(n) &= T(k) + T(n-k-1) + O(1) \\ &= T(0) + T(n-1) + O(1) \\ &= O(1) + T(n-1) + O(1) \\ &= O(1) + T(n-1). \end{aligned}$$

Solving this equation further using the iterative method, we find that $T(n) = O(n)$.
Now, we consider the preorder traversal:

```
void preorder<E>(BNode<E> node, void Function(E) fn):
    if (node != null):
        fn(node.value)
        preorder(node.left)
        preorder(node.right)
```

For the binary tree, if we start at the root with the `print` function, we get `5, 3, 8, 7, 1, 6, 2, 4`. Like the `inorder` function, this function runs in linear time.
Finally, we consider the postorder traversal:

```
void postorder<E>(BNode<E> node, void Function(E) fn):
    if (node != null):
        postorder(node.left)
        postorder(node.right)
        fn(node.value)
```

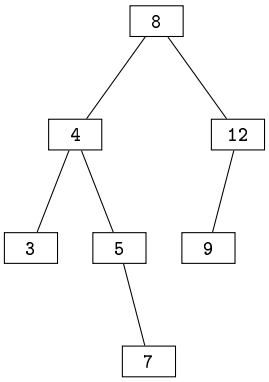
For the binary tree, if we start at the root with the `print` function, we get `8, 6, 1, 7, 3, 4, 2, 5`. Like the `inorder` and the `preorder` functions, this function runs in linear time.

Rooted trees with unbounded branching

We can extend a binary tree into a k-ary tee, storing a list of children- we just need to add all the required pointers. However, there will be a lot more `null` values as k increases. Also, we still need the value of k to be fixed.
To overcome this, we can make use of unbounded number of children. For this, we store two more attributes to a node- `leftChild`, which points to the leftmost child, and `rightSibling`, which points to the sibling of the node immediately on the right. However, accessing a child becomes an operation with running time $O(k)$ because we need to scan all the children.

Binary Search Trees

A binary search tree (BST) is a binary tree that obeys the binary search tree property. The property states that for a node `node1`, any node in the left subtree is smaller than (or equal) `node1`, while any node in the right subtree is (strictly) bigger than `node1`. For example, the following is a BST:



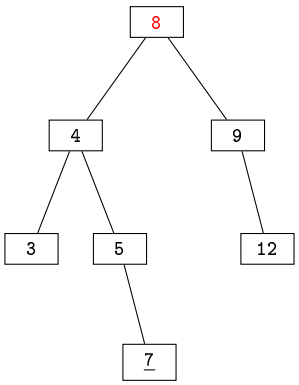
If we traverse the BST using inorder traversal, we will get a sorted sequence. In the case of the BST above, it is `3, 4, 5, 7, 8, 9, 12`.

Search

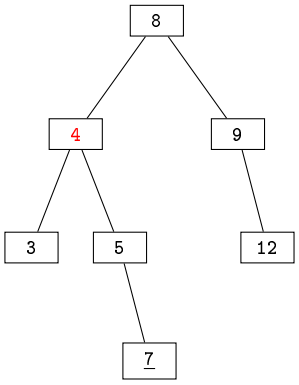
The BST has a very efficient algorithm for searching an element. We look at the pseudocode below:

```
BNode<int> search(BNode<int> node, int value):
    if (node == null || node.value == value):
        return node
    if (value < node.value):
        return search(node.left, value)
    else:
        return search(node.right, value)
```

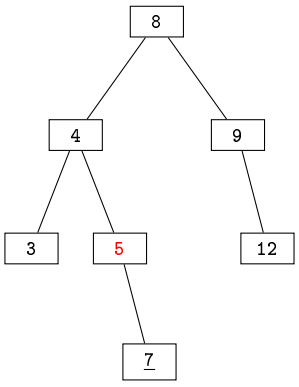
At start, we set the `node` to be the `root` of the tree. The (worst) running time of the code is $O(h)$, where h is the height of the tree. It is much more efficient than searching in a binary tree because the algorithm exploits the binary search tree property.
We now consider how the function works in the BST below when searching the value 7.



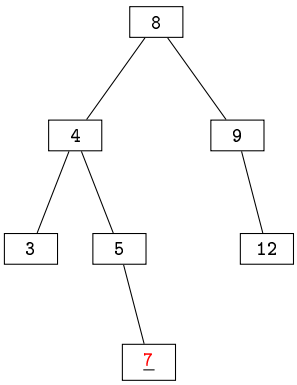
We start at the root of the tree. Since 7 is smaller than 8, we go to the left subtree.



Now, 7 is bigger than 4, so we go to the right subtree.



Since 7 is greater than 5, so we go to the right subtree again.



Since 7 is the value of this node, we return the node.
We can also define the search algorithm iteratively. This is usually more efficient because it isn't recursive.

```
BNode<int> search(BST tree, int value):
    BNode<int> node = tree.root
    while (node != null || node.value != value):
        if (value > node.key):
            node = node.left
        else:
            node = node.right
    return node
```

Maximum and minmum

Now, we define the minimum and maximum algorithms for a BST. For the minimum, we head to the leftmost node. The recursive version of the algorithm is given below:

```
BNode<int> minimum(BNode<int> node):
    if (node.left == null):
        return node
    return minimum(node.left)
```

We can also define the function iteratively:

```
BNode<int> minimum(BST tree):
    BNode<int> node = tree.root
    while (node.left != null):
        node = node.left
    return node
```

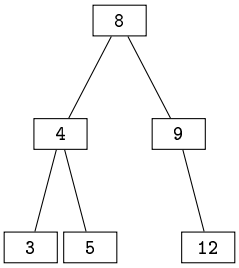
For the maximum, we head to the rightmost node. The recursive version of the algorithm is given below:

```
BNode<int> maximum(BST tree):
    BNode<int> node = tree.root
    while (node.right != null):
        node = node.right
    return node
```

Like in the case for the minimum, we can define the function iteratively:

```
BNode<int> maximum(BNode<int> node):
    if (node.right == null):
        return node
    return maximum(node.right)
```

Here, we start with the `root`. Also, there are no comparisons involved in all the procedures- we can guarantee that the value is the maximum or the minimum using the BST property. The running time of all the 4 functions is $O(h)$.
In the case of the BST



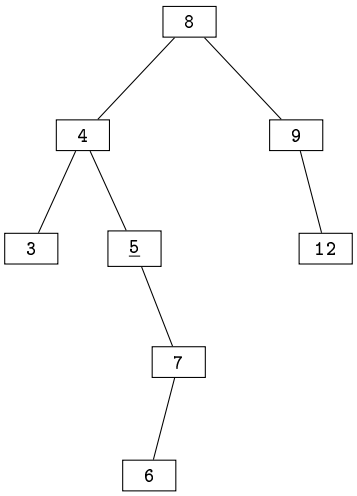
the minimum is 3, while the maximum is 12.

Successor and predecessor

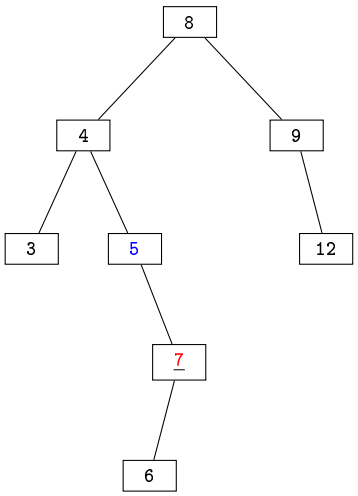
Now, we define the successor of a node. The successor is the node with the smallest value bigger than the value of the node- this is the next element in the inorder traversal. The pseudocode for successor is:

```
BNode<int> successor(BNode<int> node1):
    // if there is a right subtree, then it contains the successor
    if (node1.right != null):
        return minimum(node1.right)
    // otherwise, go up until we find a left child
    BNode<int> node2 = node1.parent
    while (node2 != null && node1 = node2.right):
        node1 = node2
        node2 = node2.parent
    return node2
```

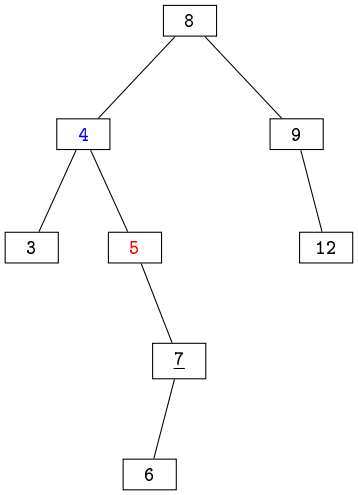
We show how we find the successor of 5 with the BST below:



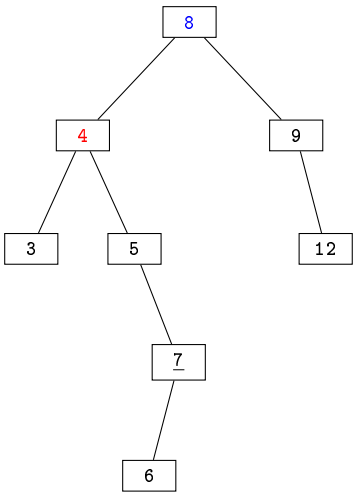
Since 5 has a right subtree, we return the leftmost element in the right subtree- this is 6.
Next, we consider finding the successor of 7.



`7` doesn't have a right subtree, so we have to enter the `while` loop. The parent is coloured blue, while the node is in red. Since `7` isn't in the left subtree of `5`, we go upwards.



Still, `5` is in the right subtree of `4`- we go up again.



Since 4 is in the left sublist of 8, we have encountered the successor of 7- it is 8.
Similarly, we can define the algorithm to find the predecessor:

```
BNode<int> predecessor(BNode<int> node1):
    if (node1.left != null):
        return maximum(node1.left)
    BNode<int> node2 = node1.parent
    while (node2 != null && node1 = node2.left):
        node1 = node2
        node2 = node2.parent
    return node2
```

Size and height

Next, we define algorithms for `size` and `height` recursively:

```
int size(BNode<int> node):
    if (node == null):
        return 0
    return size(node.left) + size(node.right) + 1
```

The size of a tree is the number of elements in the tree. The size of the tree is the size of the left subtree and the size of the right subtree summed up with an extra value of 1 for this node.

```
int height(BNode<int> node):
    if (node == null):
        return 0
    if (node.left == null && node.right == null):
        return 0
    return max(height(node.left), height(node.right)) + 1
```

The height is the maximum height of the left and the right subtree, with an extra one for the parent layer.

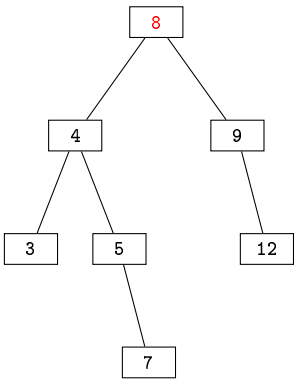
Insertion

Both these algorithms work for binary trees; the tree doesn't need to be a BST in order of the size and the height to be calculated.

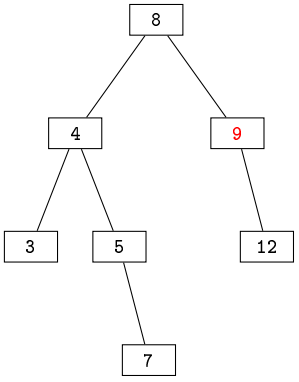
Next, we consider inserting a node into the tree. Since the BST property must be preserved, we need to ensure we add the node at the right position. The iterative version of the algorithm is:

```
void insert(BST tree, BNode<int> node1):
    // node2 is always the parent of node3
    BNode<int> node2 = null
    BNode<int> node3 = tree.root
    // make node3 the to-be parent of node1
    while (node3 != null):
        node2 = node3
        if (node1.value < node3.key):
            node3 = node3.left
        else:
            node3 = node3.right
    node1.parent = node2
    if (node2 == null):
        tree.root = node1
    else if (node1.key < node3.key):
        node2.left = node1
    else:
        node2.right = node1
```

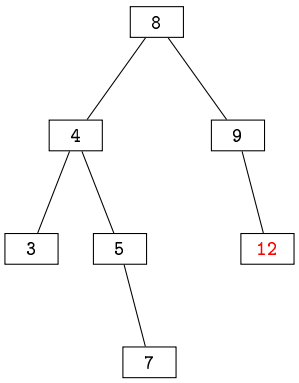
We show how the function works when adding 11 into the tree



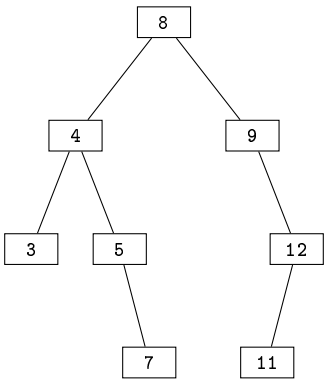
Since 8 is smaller than 11, we go to the right.



Since 11 is bigger than 9, we still go to the right.



Now, 12 is bigger than 11, so we should go to the left. However, 12 has no left subtree, so we add 11 there. Then, the tree becomes



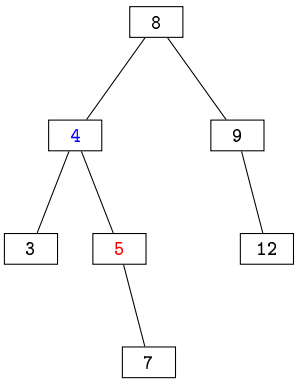
Deletion

Deletion is a bit more complicated. Deleting leaves is easy- we make the parent not have a left/right subtree. Also, deleting a node with only one subtree is easy- we connect the parent of the node to the child. It is however not trivial in the case that the node has two children. There are many possibilities here.

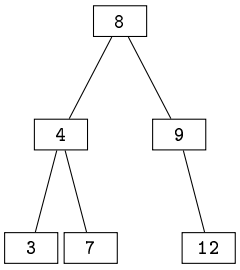
First, we introduce the algorithm `transplant`- this replaces the subtree rooted at `node1` with the subtree rooted at `node2`.

```
void transplant(BST tree, BNode<int> node1, BNode<int> node2):
    if (node1.parent == null):
        tree.root = node2
    else if (node1 = node1.parent.left):
        node1.parent.left = node2
    else:
        node1.parent.right = node2
    if (node2 != null):
        node2.parent = node1.parent
```

We show how the algorithm works by transplanting 4 and 5 in the following BST:



Since 5 is the right child of 4, we make 7 the right child of 4. Also, we make 4 the parent of 7. So, 5 is no longer in the tree, and the tree becomes



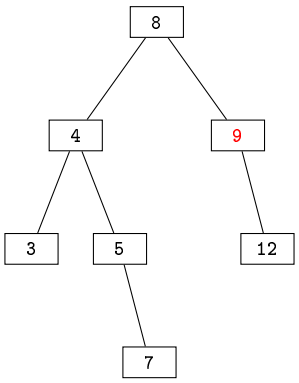
Note that the BST property is still satisfied.

Now, we define the delete function:

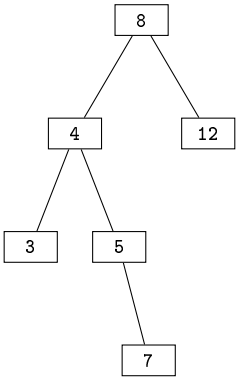
```
void delete(BST tree, BNode<int> node):
    if (node.left == null):
        transplant(tree, node, node.right)
    else if (node.right == null):
        transplant(tree, node, node.left)
    else:
        BNode<int> node1 = minimum(node.right)
        if (node1.parent != node):
            transplant(tree, node1, node1.right)
            node1.right = node.right
            node1.right.parent = node1
        transplant(tree, node, node1)
        node1.left = node.left
        node1.left.parent = node1
```

It runs in $O(h)$ time because we call the `minimum` function. The `transplant` function runs in constant time.

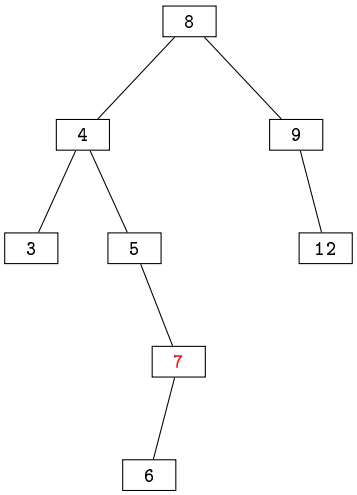
We now consider the 4 cases for deletion. First, we delete the node 9 from the tree



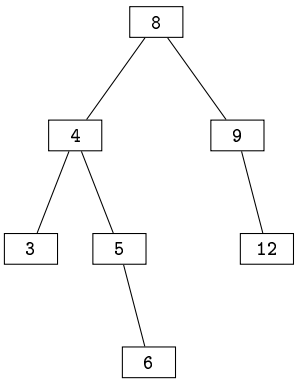
Since 9 doesn't have a left subtree, we perform a left transplant. Then, the BST becomes:



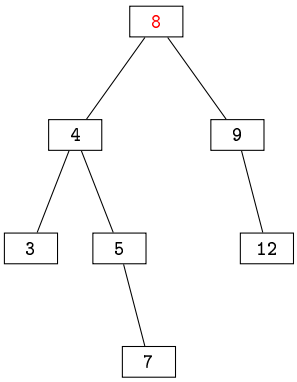
Next, we consider removing 7 from the tree



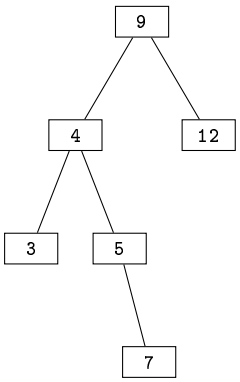
Since 7 doesn't have a right subtree, we perform a right transplant. Then, the BST becomes:



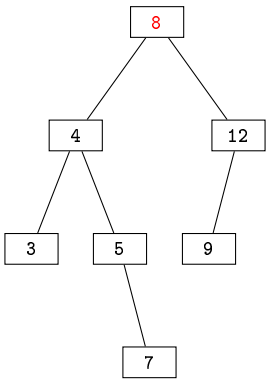
Now, we consider deleting the node 8 from the BST



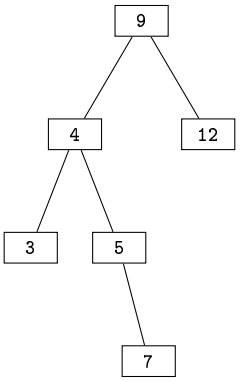
Since 8 has both a left and a right child, we find the minimum element in the right. This is 9. Since 9 is the right child of 8, we make it replace 9. So, the tree becomes:



Finally, we consider deleting 8 from the BST

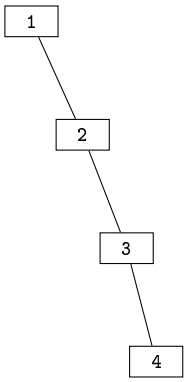


Since 8 has both a left and a right child, we find the minimum element in the right. This is 9, but 9 isn't the right child of 8. So, we first replace 9 with its parent 12. Then, we make 9 replace 8. So, the tree at the end is

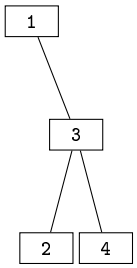


Balanced and unbalanced BSTs

To exploit the binary tree structure, we need to ensure that the tree is balanced. If not, then we will have $h = n$, so the functions with running time $O(h)$ are linear. If the tree is balanced, then we have $O(h) = O(\log n)$, which is much more efficient. So, there are many strategies to a BST to ensure it remains balanced. When randomly inserting to a tree, the height is $O(\log n)$. A common method to ensure this is by making use of rotations. For example, in the BST below



we can left-rotate on 2 to get



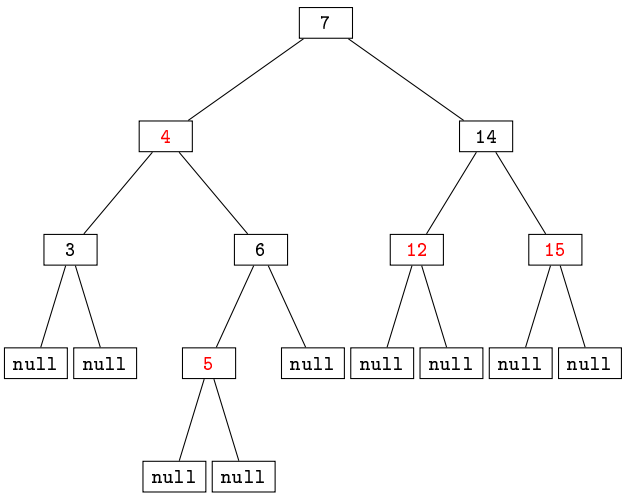
Rotations do not affect the BST property. Here, we only update pointers, so it runs in constant time.
There are many ways of storing duplicate keys in a BST- we can keep the nodes separate; keep a count of the nodes; add them randomly to left or the right; or ignore them completely. If there are a lot of duplicates, then there it is more likely for a tree to be unbalanced.

Red-Black Trees

A red-black tree is a binary tree that is self-balancing. A red-black tree node has an extra attribute colour, which is either red or black. Also, For every node with value in the tree, we attach 2 `null` elements to it so that it is not a leaf. Moreover, the following 5 properties are obeyed:

1. Every node is either red or black;
2. The root is black;
3. Every leaf (i.e. the `null` nodes) is black;
4. If a node is red, then both its children are black;
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

For example, the following is a red-black tree:



All 5 properties are obeyed by the tree here- a node is either red or black; the root is black; the leaves are black; a red parent has black children; and for each node, the black height of all the paths is the same.

Instead of storing all the `null` nodes, we make use of a sentinel- the two children of a leaf are the sentinel, along with the parent of the root. This is an optimisation. Typically, we represent the tree without its null nodes, but the nodes are still there in the implementation.

Satisfying the red-black property implies that the height of the tree is at most $2\log(n+1)$. First, we show that the subtree rooted at the node `x` contains at least $2^{\text{bh}(x)} - 1$. We do this by induction.

- In the base case, $\text{bh}(x) = 0$. So, the node `x` is a sentinel. Moreover, its subtree contains $2^{\text{bh}(x)} - 1 = 2^0 - 1$ nodes.

- Now, if we have an internal node `x` with a positive height and two children. Either its child has black height $\text{bh}(x)$ (i.e. the child node is red), or $\text{bh}(x) - 1$ (i.e. the child node is black). Using the inductive hypothesis, we conclude that the subtree rooted at a child has at least $2^{\text{bh}(x)-1} - 1$ internal nodes. So, the minimum number of internal nodes of `x` is

$$(2^{\text{bh}(x)-1} - 1) + (2^{\text{bh}(x)-1} - 1) + 1 = 2^{\text{bh}(x)} - 1.$$

Next, we show that the height of the tree $h \leq 2\log(n+1)$. By Property 4, we know that at least half of the nodes on a path are black. So, $\text{bh}(x) \geq h/2$. Then,

$$n \geq 2^{h/2} - 1 \implies n + 1 \geq 2^{h/2} \implies \log(n+1) \geq h/2 \implies h \leq 2\log(n+1).$$

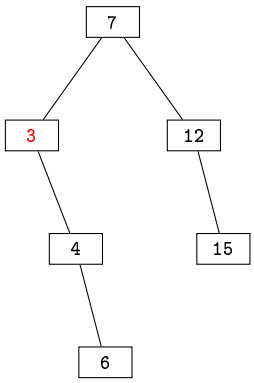
So, the operations on a red-black tree take $O(\log n)$ in the worst case.

When inserting or deleting a node from a red-black tree, the red-black property might be violated. So, we make use of left- and right-rotations to to restore the property. The pseudocode for left-rotation is given below:

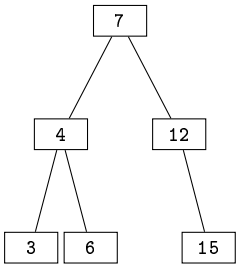
```
void leftRotation(BST tree, BNode<int> node1):
    BNode<int> node2 = node1.right
    node1.right = node2.left
    if (node1.left != null):
        node1.left.parent = x
    node2.parent = node1.parent
    if (node1.parent == null):
        tree.root = node2
    else if (node1 = node1.parent.left):
        node1.parent.left = node2
    else:
        node1.parent.right = node2
    node2.left = node1
    node1.parent = node2
```

Since the operations involved are just pointer manipulations, the function runs in constant time. The right rotation algorithm is symmetric to the left rotation.

We show how the algorithm using the BST by rotating left around 3.



By rotating left, we make 4 the parent of 3 and the left child of 7.



Since 3 had no left child, we did not need to make any further changes to the structure of the tree.

Insertion

Using this function, we define the insertion function.

```
void insert(BST tree, BNode<int> node1):
    BNode<int> node2 = null
    BNode<int> node3 = tree.root
    while (node3 != null):
        node2 = node3
        if (node1.value < node3.key):
            node3 = node3.left
        else:
            node3 = node3.right
    node1.parent = node3
    if (node2 == null):
        tree.root = node1
    else if (node1.key < node3.key):
        node2.left = node1
    else:
        node2.right = node1
    node1.left = null
    node1.right = null
    node1.colour = Color.red
    fixup(tree, node1)
```

This function is very similar to the insertion for a BST, just with few extra lines at the end. First, we make the node red. Then, we make use of the function `fixup` that restores the red-black property.

Now, we consider the `fixup`. After adding a new node, we still satisfy properties 1 (a node is red or black), 3 (leaves are black), and 5 (black height is always equal). If the added node was the root, then the root might not have been black. Moreover, if the node's parent might have been red. The function `fixup` aims to ensure the two properties are still satisfied. We now define the function `fixup`.

```
void fixup(BST tree, BNode<int> node1):
    while (node1.parent.colour == Color.red):
        if (node1.parent == node1.parent.parent.left):
            BNode<int> node2 = node1.parent.parent.right
            // case 1
            if (node2.colour = Colour.red):
                node1.parent.colour = Colour.black
                node2.colour = Colour.black
                node1.parent.parent.colour = Colour.red
                node1 = node1.parent.parent
            else:
                // case 2
                if (node1 = node1.parent.right):
                    node1 = node1.parent
                    leftRotation(tree, node1)
                // case 3
                node1.parent.colour = Colour.black
                node1.parent.parent.colour = Colour.red
                rightRotation(tree, node1.parent.parent)
        else:
            // cases 4, 5, 6 which are symmetric to cases 1, 2, 3
        tree.root.colour = Colour.black
```

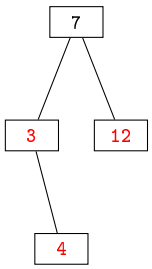
We need to ensure properties 2 and 4. The last line ensures that property 2 is satisfied- the colour of the root is black. The `while` loop ensures that there are no two parent-child nodes with both red colour. There are 6 possible cases in the while loop, 3 for the parent being a left child, and 3 for the parent being a right child. The 3 cases for the parent being a left child is:

- If the uncle node of the added node is red, then we "push down the blackness". That is, we make the parent and the uncle black, and make the grandparent node red. We might still have a violation of property 4, so we make `node1` the grandparent node and continue the `while` loop.

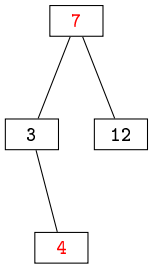
- If the uncle node is black and `node1` was the right child, then we need to rotate to the left the node. This converts fixup case 2 into fixup case 3.

- If the uncle node is black and `node1` is the left child, then we make the parent black and the grandparent red. Then, we rotate the grandparent to the right. The while loop terminates here since the parent becomes black, and the children are both black.

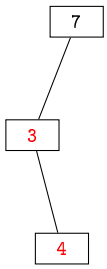
We now consider the 3 left cases of fixup in detail below. Suppose we have added 4 into the tree, and the BST has become:



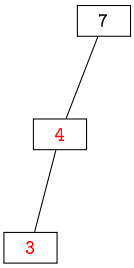
We are at case 1- both the parent and the uncle are red. So, we push down the blackness from the top. So, 3 and 12 become black, while 4 stays red. The BST is now:



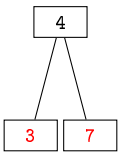
We have made no rotations in this case. We can then go upwards and check whether 7 violates property 4.
Next, we consider adding 4 into the tree, making the BST:



We are at case 2- the uncle is black (`null` nodes are black). Since 4 is a right child, we perform a left rotation on 4. Then, the tree becomes:



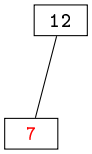
Now, we're at case 3. We rotate 7 to the right, making 4 the parent of both 3 and 7.



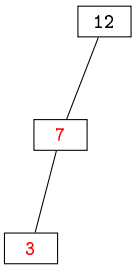
Here, the parent is black, and the children are red. There is no possible violation here, so the while loop terminates.
We reconsider the insertion and the fixup by adding the values `12, 7, 3, 15, 4, 6, 14, 5` into an empty red-black tree. We first add `12` to the node.

Since the root must be black, after the fixup, we end up with

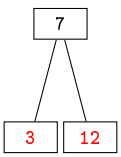
Next, we add `7` to the tree.



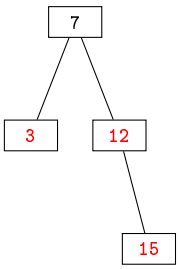
There is no violation of red-black property here, so the fixup procedure leaves the tree as is. Then, we add `3` to the BST:



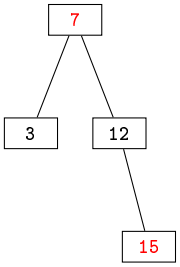
Here, we have a violation of property 4- both `7` and its child `3` are red. Since the uncle of `3` is black, and `3` is the left child of `7`, we are in case 3. So, `7` becomes the root, its colour changes to black, and `12` also becomes red. Then, the BST becomes:



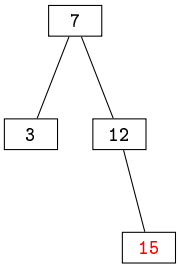
Now, we add `15` to the tree.



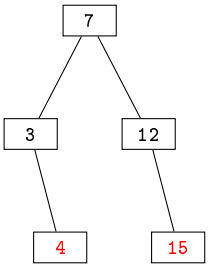
We have another violation of property 4- both 12 and 15 are black. Since the uncle of 15 is red, we need to push the blackness down. The tree is now:



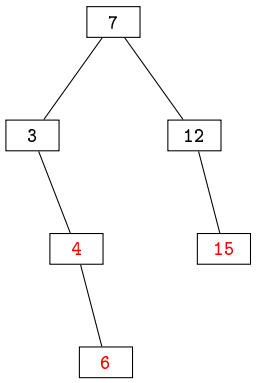
Since property 2 is not satisfied, we make 7 black. So, the BST is



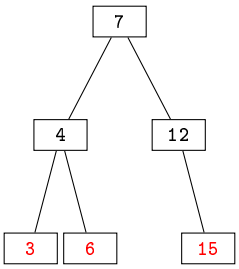
The next node to add is 4.



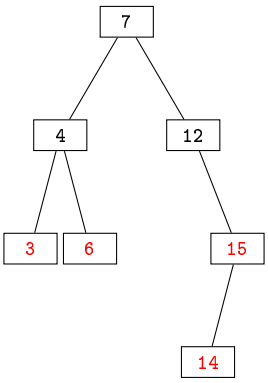
There is no violation of the red-black property here. Next, we add 6.



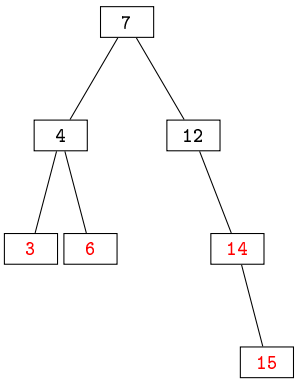
Here, we violate property 4. We are in the symmetric case of 3. So, we left rotate on 3, along with inversion of the colours for 3 and 4. The BST is now



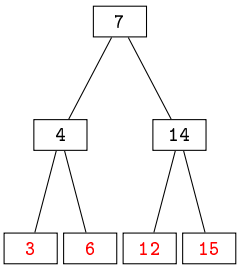
Then, we add 14 to the BST:



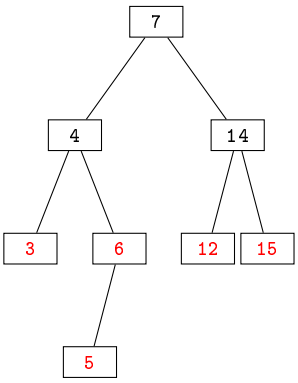
We have another violation of property 4 now. We are in the symmetric case of case 2. First, we rotate to the right on 14.



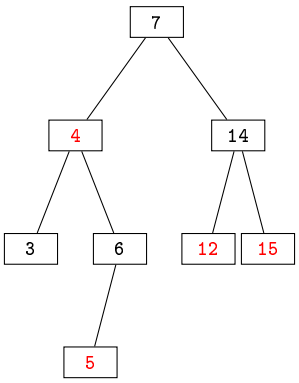
We are now in the symmetric case of 3. So, we left rotate on 12, along with inverting the colours of 12 and 14. The BST is therefore:



Finally, we add 5 to the tree.



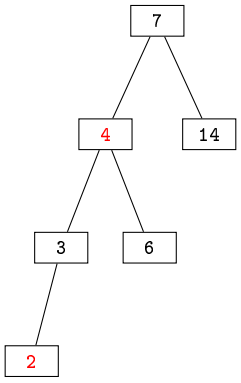
Here, we have a violation of red-black property. We are in case 1. So, we push the blackness down.



There is no further violation, so this is the final state of the BST.
The normal BST insertion takes $O(h) = O(\log n)$ running time. In the worst case of the fixup, we need to continue the while loop until the root, i.e. the running time is $O(\log n)$. There is also at most 2 rotations performed per insertion of a node, since rotations terminate the while loop.

AVL trees

AVL trees are another class of BST that aims to keep the tree balanced. It obeys a much stricter rule- for every node x, the difference in the heights of the left and the right subtree is at most 1. We keep track of this by the help of another attribute to the node- the height.
Every AVL tree is a red-black tree. But, not every red-black tree that is a valid AVL tree. For example, the tree



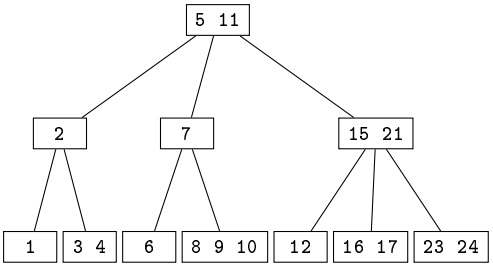
is a valid red-black tree, but not a valid AVL tree. In particular, the difference in the height of the left and the right subtrees of 7 is 2.
Because red-black trees make much fewer rotations than AVL trees, it is more efficient when adding or removing an element. However, AVL is a bit more efficient at searching. In fact, for an AVL, $h \leq 1.44 \log n$. As a result, red-black trees are used in real-time applications and most implementation of ADTs like set and map, while AVL trees are used in databases.

B-trees

B-trees are an extension to binary search trees, and are used to store a large amount of data. It is used for storing data in disks or other secondary storage devices. It minimises I/O operations. Unlike BST, a B-tree can store more than 2 children. The height is $O(\log n)$, but it is much lower than that of a red-black tree.
A b-tree node has the following attributes:

- `length`, which keeps track of the number of keys in the node;
- `key`, which keeps an array of keys stored in a non-decreasing order;
- `isLeaf`, which checks whether the node is a leaf;
- `children`, which stores an array of pointers to the $n+1$ children of the node.

A b-tree has a `root` attribute which points to the root of the tree. We normally fix the number of children a node can have. An example of a b-tree is:



We say that the B-tree above has degree $t = 2$. Moreover, the following are the properties of a b-tree:

- the leaves have no children;
- the keys separate the ranges stored in each subtree. For example, in the rightmost branch of the tree (`15 21`), we have only `12` in the leftmost child since it is smaller than `15` (but still bigger than `11`), the middle children have value between `15` and `21`, while the rightmost children are bigger than `21`;
- All the leaves have the same depth- this is the height of the tree;
- Every node other than the root must have at least $t-1$ keys;
- A node can only contain up to $2t-1$ keys. If a node contains $2t-1$ keys, then it is a full node. Here, `8 9 10` is a full node.

We now find a bound for the worst-case height of a B-tree. In fact, if we have $n \geq 1$ elements in a B-tree of degree $t \geq 2$, then the height is bounded by

$$h \leq \log_t \frac{n+1}{2}.$$

We know the root of the B-tree has at least one key, and all the other nodes have at least $t-1$ keys. So, the B-tree has at least 2 nodes at depth 1, $2t$ nodes

at depth 2, $2t^2$ nodes at depth 3, and so on. In general, this is $2t^{h-1}$ nodes at depth h. So,

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) = 2t^h - 1.$$

Now, we can rearrange this expression to find

$$h \leq \log_t \frac{n+1}{2}.$$

If we have a B-tree with branching factor 1001 and height 2, we can store

$$1 \cdot 1000 + 1001 \cdot 1000 + 1001^2 \cdot 100 \geq 1 \times 10^9$$

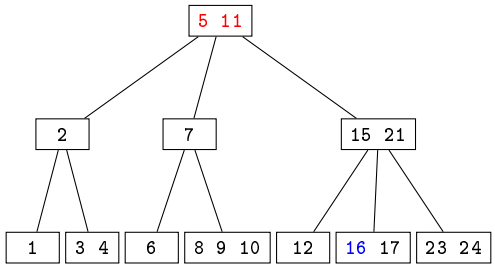
keys. This allows for very efficient access to a key when we are retrieving something from a huge pool of data, e.g. disk accesses.

Search

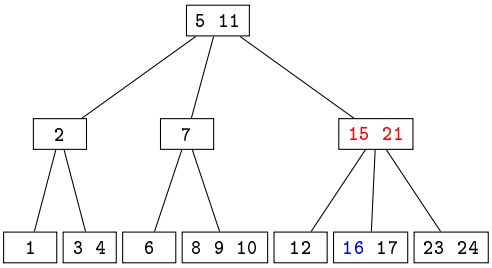
We now consider the operation of searching a B-tree for content. Since this is not a common structure used in everyday programming, we consider this in the case of searching within a B-tree with data from a disk. For this, we define the three constant operations: `readDisk`, which reads an object into a memory; `writeDisk`, which saves the object onto disk; and `allocateNode`, which allocates a disk page. We also expect the root to be stored in main memory.
The pseudocode for the `search` operation is:

```
[Node<int>, int] search(Node<int> node, int value):
    int i = 0
    while (i < node.length && k > node.value[i]):
        i++
    if (i < node.length && k == node.value[i]):
        return [node, i]
    else if (node.isLeaf):
        return null
    else:
        readDisk(node.children[i])
        return search(node.children[i], k)
```

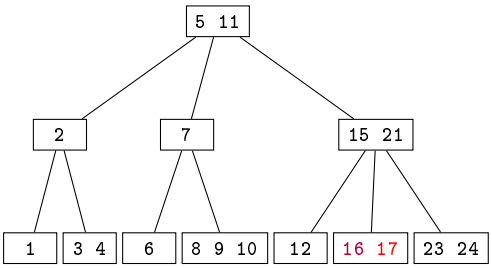
This is a recursive algorithm and is very similar to the search algorithm for a BST. We start the algorithm with the root. The running time of the algorithm is $O(\log_t n)$, and the total CPU time is $O(t \log_t n)$ since the while loop takes $O(t)$ (the values within the node is at most $2t$).
We illustrate the algorithm by finding `16` in the following B-tree.



We start checking from the root. Since $16 > 11$, we need to go down the right child, i.e. we disk read that child and then recurse on it.



Since $16 > 15$ but $16 < 21$, we go down the middle child.



The first element here is 16, so we have found the node with that value. We return the node `16 17` and index `0`.

Create a B-tree

The following is the pseudocode for creating an empty B-tree:

```
void create(Tree tree):
    Node node = allocateNode()
    node.isLeaf = true
    node.length = 0
    writeDisk(node)
    tree.root = node
```

This takes constant disk operations and constant CPU time.

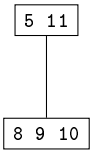
Insertion

Like with a BST, when inserting to a B-tree, we find the leaf where we can add the key. However, insertion might violate the B-tree property. For example, we might be adding to a full node. In that case, we need to split child. In that case, we push the median element to the parent, and split the child into two branches. Of course, it is possible for the parent to now exceed its capacity- we need to continue this procedure with the parent. The following is the procedure for splitting child:

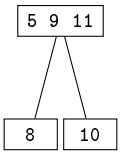
```
void splitChild(Node<int> node1, int i):
    Node<int> node2 = allocateNode()
    Node<int> node3 = node1.children[i]
    node2.isLeaf = node3.isLeaf
    // t is the degree of the B-tree
    node2.length = t-1
```

```
    for (int j = 0; j < t-1; j++):
        node2.value[j] = node3.value[j+t]
    if (!node3.isLeaf):
        for (int j = 0; j < t; j++):
            node2.children[j] = node3.children[j + t]
    node3.length = t-1
    for (j = x.length; j > i; j--):
        node2.children[j+1] = node2.children[j]
    x.children[i+1] = node2
    for (j = node1.length; j >= i; j--):
        node1.value[j+1] = node1.value[j]
    node1.value[i] = node3.value[t]
    node1.length = node1.length+1
    writeDisk(node3)
    writeDisk(node2)
    writeDisk(node2)
```

Here, `node1` is the parent with a full child node at index `i`. For example, if the B-tree has the following branch:



we can split the child to get the following branch.



The running time of the algorithm is $O(t)$. There are only constant number of disk operations performed. After splitting the child, the height can grow by one.
Now, we define the insertion procedure for a non-full node.

```
void insertNonfull(Node<int> node, int value):
    int i = node.length-1
    if (node.isLeaf):
        while (i >= 0 && value < node.value[i]):
            node.value[i+1] = node.value[i]
            i--
        node.value[i+1] = k
        node.length = node.length+1
    else:
        while (i >= 0 && value < node.value[i]):
            i++
        i++
        readDisk(node.children[i])
        if (node.children[i].length == 2t-1):
            splitChild(node, i)
            if (value > node.value[i]):
                i++
        insertNonfull(node.children[i], value)
```

There are 2 cases, depending on whether the node is a leaf (base case), or if it isn't a leaf. Note that we readily split the child when a non-leaf node is full.
Now, we define the insertion algorithm:

```
void insert(Tree tree, int value):
    Node<int> root = tree.root
    if (root.length == 2t-1):
        Node<int> newRoot = allocateNode()
        tree.root = newRoot
        newRoot.isLeaf = false
        newRoot.length = 0
        newRoot.children[0] = root
        splitChild(newRoot, 0)
        insertNonfull(newRoot, value)
    else:
        insertNonfull(root, value)
```

If the root is full, then we need to split it before calling `insertNonfull` on the new root. The algorithm uses CPU $O(t \log_t n)$, while it uses $O(\log_t n)$ disk accesses.

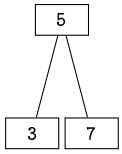
We now consider adding a sequence of keys into an empty B-tree of degree $t = 2$. Since $t = 2$, we can store a maximum of 3 values per node.

First, we add 5 to the tree.

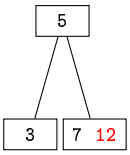
Next, we add 7 to the tree.

Since $7 > 5$, we add it to the right of 5. Now, we add 3.

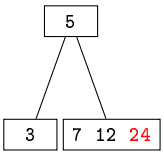
The root node is now full, so if we want to add the element 12, we need to first split.



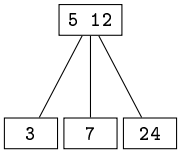
The split pushes the median element to the top, and places those to the left of the median in the left child, while those to the right in the right child. Then, we can actually add 12.



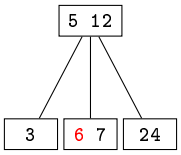
Since $12 > 5$, we add it to the right leaf node. Now, we add 24.



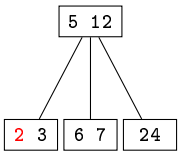
Here, the right child of the root is a full node. So, to add the next element 6 in that node, we need to split it.



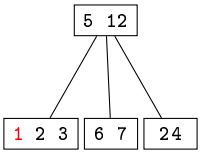
When we split, we push the median 12 to the root, making 7 and 24 the middle and the right children of the root respectively. Now, we can add 6.



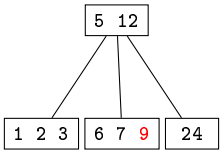
Since $6 > 5$ and $6 < 12$, it gets added to the middle child of the root. Next, we add 2.



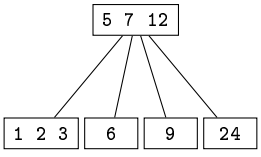
Then, we add 1.



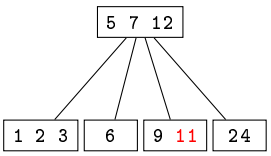
Now, we add 9.



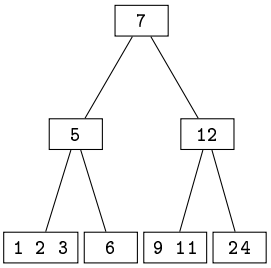
Now, we want to add 11. First, we need to split the middle child of the root since it is a full node.



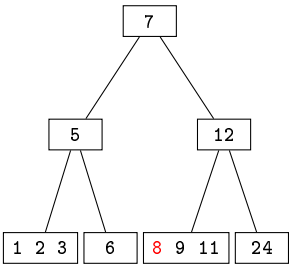
Now, we can add 11.



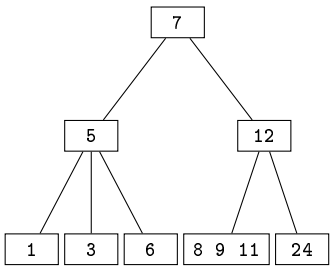
Since the root (a non-leaf node) is full, we split the node when we add the next element 8.



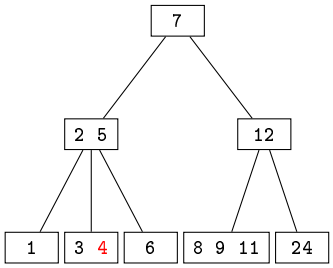
Now, we add 8.



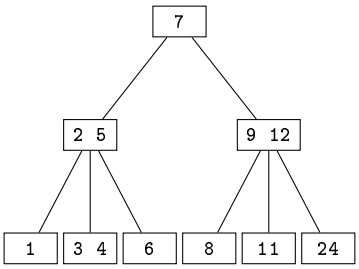
Now, we want to add 4. Since it will go to the leftmost node, and the node is full, we need to split it, promoting 2 to the parent.



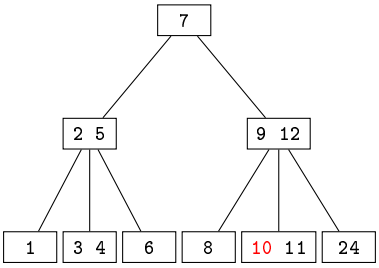
Then, we add 4.



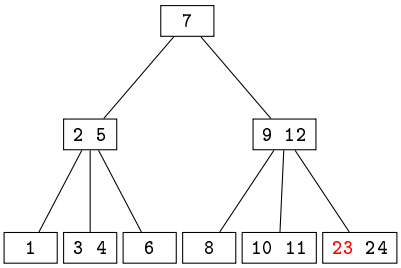
Now, we want to add 10. This would go to the node 8 9 11, so we first split.



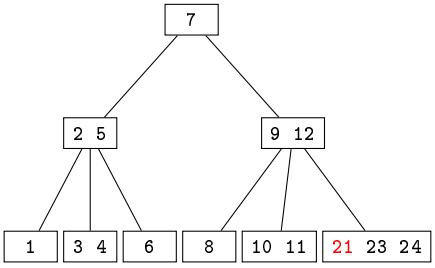
Then, we add 10.



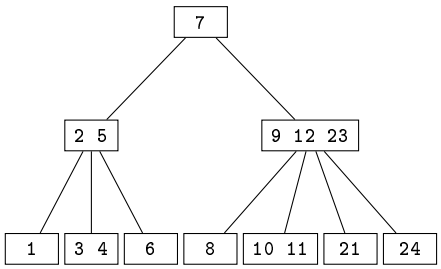
Now, we add 23.



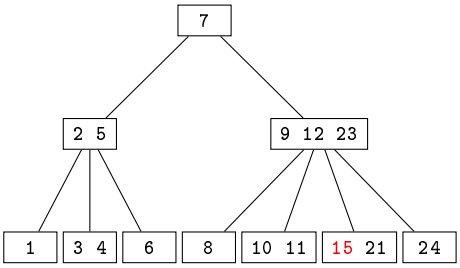
Then, we add 21.



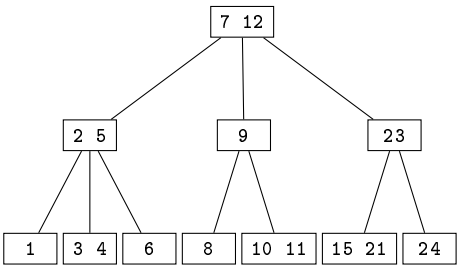
Next, we add 15. But, this goes to the rigthmost leaf, meaning that we first split.



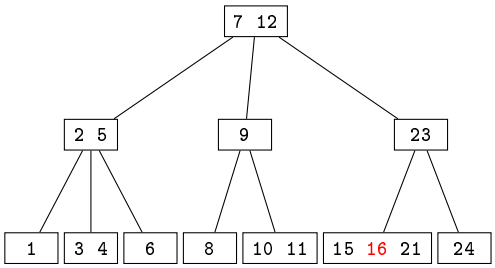
After the split, we add 15.



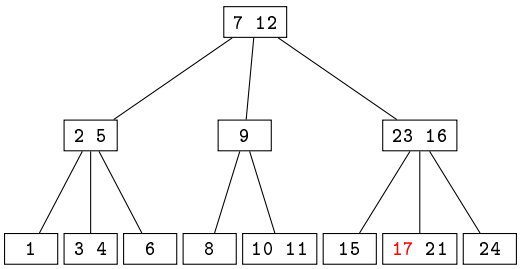
Since 9 12 23 is a full non-leaf node, when we add the next element 16, we first split that node.



Then, we add 16.



Finally, we add 17.



Variants to the B-tree

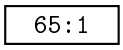
We now consider some variants to the B-tree. A 2-3 tree is a type of B-tree where every internal node has either two or three children. We can similarly define a 2-3-4 children. A B+ tree stores all the satellite information in the leaves; only keys and child pointers are in the internal nodes. B* trees try to have more densely packed, i.e. it ensures that non-root nodes are at least 2/3 full, instead of the 1/2 in a normal B-tree.

2.4 Maps

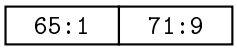
The map abstract data type contains a collection of key-value pairs. Often, we don't allow duplicate keys. The main operations defined on a map are:

- `void insert<K, V>(Map<K, V> map, K key, V value)`, which inserts the pair `(key, value)` into the map.
- `V delete<K, V>(Map<K, V> map, K key`, which removes the entry with key `key` from the map and returns its `value`.
- `V search<K, V>(Map<K, V> map, K key)`, which returns the `value` of the entry with key `key` without removing it.

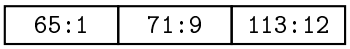
We also have the operation `bool isEmpty<K, V>(Map<K, V> map)`, which returns `true` only if the map has no entries.
We now consider how the map ADT stores the content. We start with an empty map. To it, we add the entry `(65, 1)`.



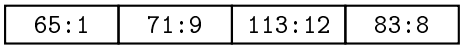
Here, `65` is the key, while `1` is the value. The size of the map is 1. Next, we add the entry `(71, 9)`:



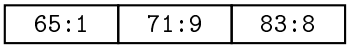
Since `71` isn;t in the list, we add it Now, we add the entry `(113, 12)`:



If we now search for `65`, we will get back `1`. The map remains unchanged. Then, we can insert the entry `(83, 8)`:



Next, we delete `113`. So, we get back `12` and the map is now



Finally, if we search for `113`, we get `null`. The map remains unchanged.

List-based maps

We can use a doubly linked list to implement a map. Here, we store keys as the value of a node, along with an attribute pointing to the value. We know the complexities of the different operations on a linked list. In particular:

- `insert` takes constant time. If we checked for duplicates, it would take linear time. However, this isn't necessary as the most up-to-date value (if any) will be closest to the head.
- `search` and `delete` operations take linear time. This is because, at the worst case, we need to traverse the entire list.

Since maps are a very commonly used data structure, we can only make use of this implementation when storing maps of small size.

Tree-based maps

If we instead use a map based on self-balancing trees, then the complexity of `insert`, `search` and `delete` becomes $O(\log n)$. Moreover, traversing inorder gives us a sorted sequence of entries with respect to the key. Here, like in the case of a linked list, we store the value as an attribute of the key. This is a better implementation than the linked list version, but the operations are still not constant.

Direct-address tables

Direct address tables is a data structure similar to the way an array can be used to implement maps with integer keys. The complexities of the 3 operations is $O(1)$ since it is equivalent to array indexing and rewriting.
In a direct-address table, the keys are from a universal set containing integers from 0 to m, where m is not very big. If we have an entry `(i, value)` in a map, then in the table, we expect `value` to be in the `i`-th position of the table. If there is no entry with key `j`, then the `j`-th position of the table should be `null`.
Using array indexing, the pseudocode for the operations are:

```
void insert<K, V>(Map<K, V> map, MapEntry<K, V> entry):
    map[entry.key] = entry.value
```

```
V search<K, V>(Map<K, V> map, K key):
    return map[key]
```

```
V delete<K, V>(Map<K, V> map, K key):
    value = map[key]
    map[key] = null
    return value
```

The limitation comes from the size of the array- we will have a lot of unused places if the size is too big. Also, the keys must be integers, which is another big limitation to this.

Hash tables

To overcome the two issues, we make use of a hash table. We store a collection of entries by the following two main steps:

- We convert the key of an entry into an integer between 0 and a size less than a fixed integer m. This is done by the hash function.
- We save the hash table of size $m+1$ with the values in the relevant index, like in a direct-address table.

The hash function is expected to run in constant time. So, the operations `insert`, `delete` and `search` run in constant time on average.
We expect the integer m to be much lower than the possible values of the keys (which might even be infinite, e.g. strings). So, we will have hash collisions- two keys can be mapped to the same integer by the hash function. Ideally, the number of hash collisions should be minimised by the hash function. Nonetheless, hash collisions can't be avoided.

We now consider implementing the map earlier described using a hash table. We choose the hash function $h(k) = k \bmod 8$, with $m = 7$.

null	null	null	null	null	null	null	null

Initially, we have an empty table with `null` in each of the 8 positions. First, we add the entry `(65, 1)`.

null	65:1	null	null	null	null	null	null

We have $h(65) = 1$, so we add the value `1` at index `1`. Next, we add `(71, 9)`.

null	65:1	null	null	null	null	null	71:9

Since $h(71) = 7$, we add the value at index `7`. Now, we add `(113, 12)`.

null	113:12	null	null	null	null	null	71:9

Since $h(113) = 1$, we add the value `12` at index `1`. This overrides the previous entry `(65, 1)`- we will look later at how this collision is resolved. For now, we will just remove the entry from the table. Then, we add `(83, 8)`.

null	113:12	null	83:8	null	null	null	71:9

Since $h(83) = 3$, we add it at index `3`. Nothing gets overridden here.
We now look at the methods of converting a value into an integer using different methods:

- The simplest way to create the integer is by taking the memory address of the key. This is the default hash code used in many programming languages. It is good for most objects, but doesn't work as well for numbers and strings.
- We can cast an object into an integer if possible. For example, in Java, we can convert the types `byte`, `short`, `char` and `float` into `int`. It is however not possible to use it for `long` and `double` since these are supersets of integers.
- We can use component sum in the case of `long` and `double`. We first partition the bits of the key into components of fixed length (e.g. 16 or 32 bits), and then sum them to get an integer- any overflow is ignored.
- Another way to hash strings is polynomial accumulation. First, we partition the bits of the key into components of fixed length- let the partitions be $a_0, a_1, \cdots, a_n$. Then, we fix an integer z and set the hash function to be
$$p(z) = a_0 + a_1 z + a_2 z^2 + \cdots + a_n z^n.$$

There might be an overflow if the key is very long. This works very well for strings. For example, if we want to convert the string `'tmp'` into an integer using their ASCII code (`t:116, m:109, p:112`), then we have

$$p(z) = 116 + 109z + 112z^2.$$

Now, we can select a value for z to find the hash value.

Hash functions

We want the hash function to be something that is easy to compute, while ensuring that every index has an equal probability to be chosen given a random set of keys. We assume that our keys are integers from now.
A possible function is truncation- we take the first few or the last few digits. It is common when we consider the id of an entity that a lot of digits are similar. In those cases, we can only consider the ones that change most often. Otherwise, we can end up with a lot of collisions. This is very easy to compute, but not always the most efficient.
Another thing we can do is division. Here, we take the remainder of number when we divide it by another fixed number, i.e. modular division. This is very fast, but we need to avoid choosing the fixed number being a power of 2- that would be equivalent to truncation. To ensure a uniform distribution, we typically fix the number to be a prime which is not very close to a power of 2. For example, if we expect to hold 5000 keys, we will choose a prime close to 5000, but not too close to a power of 2. $2^{12} = 4069, 2^{13} = 8192$, so we can choose the prime 5003. The hash function is therefore $h(k) = k \bmod 5003$.
Next, we look at the multiplication method. This method consists of two steps:

- Multiply the key `key` by a constant `A` between 0 and 1 (exclusive), and take the fractional part of `key*A`.
- Multiply the fraction by another fixed constant `m`, and take the floor of the value.

In other words,$h(k) - \text{floor}(m\,\text{frac}(kA))$. By empirical studies, we know that choosing m to be a power of 2, and A to be the conjugate of the golden ratio:

$$A = \frac{\sqrt{5}-1}{2},$$

the hash function is the most efficient. If the word size of a machine is `w` bits, and `k` fits into a single word, then we let `A` be in the form $s/2^w$, where $0 < s < 2^w$. After the multiplication, $k{*}s$ is a fixed point 2w-bit value $r_1 2^w + r_0$. We just want the most significant bits of r_0.
For example, let the key $k = 3278$, and we choose $p = 11$, $m = 2^{11} = 2048$ and $w = 32$. We find A to be $s/2^w$ closest to the conjugate of the golden ratio. In this case, we get $s = 265443769$- this value is typically stored within an architecture. After the multiplication $k * s$, we get 8701240450782. If we split this into 32-bits, we get

$$(2025 * 2^{32}) + 3931676382,$$

so $r_1 = 2025$ and $r_0 = 3931676382$. We now take the 11 most significant bits of r_0. So, the hash value is

$$(r_0 - (r_0 \bmod 2^{w-p}))/2^{w-p} = (r_0 - (r_0 \bmod 2^{21})) = 1874.$$

Universal Hashing

If we make use of a fixed hash function for all computations, then we can easily produce a set of integers that produce the same hash code. Such a system is vulnerable to malicious attacks. If we make use of universal hashing, then we randomly pick a hash function from a set of functions. This means that there is no one input that the system cannot handle.
If H is a set of hash functions for keys between 0 and $m-1$. Then, this is a universal set of hash functions if for two distinct keys k, l between 0 and $m-1$, the number of functions $h \in H$ with $h(k) = h(l)$ is at most $|H|/m$.
To create such a set, we first fix a large prime p that is bigger than all the keys. Then, we define the hash function h_{ab}, for $a \in \mathbb{Z}_p, b \in \mathbb{Z}_p^*$ such that

$$h_{ab} = ((ak + b) \bmod p) \bmod m.$$

We define the universal set as $H_{pm} = \{h_{ab} \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$. The value of $m < p$ doesn't matter. The set H_{pm} contains $p(p-1)$ hash functions since there are p choices for a and $p-1$ choices for b.
There are other advanced hashing functions as well. Cyclic redundancy checks (CRC) work better for longer keys, but can be about 3-4 times slower than the multiplicative hashing function. Cryptographic hashing functions are functions such that it is infeasible to invert the hashing, e.g. MD5 and SHA-1. MD5, the simplest cyptographic hashing, takes about double the time needed for CRC hashing. We can pre-compute hash codes to take less time when calculating.

Collision Resolution

Now, we look at two methods of collision resolution: chaining and probing.

When we resolve collision by chaining, we store all the elements in the same slot, but we chain all the elements together, i.e. store it in a doubly linked list. This ensures fast insertion and deletion. We can store the head entry in the hash table, or add a pointer to the head. For efficiency, we need to ensure that the chains are short; otherwise, the implementation becomes identical to the normal linked list version.
The operations of `insert`, `search` and `delete` for such a map is given by adding/removing from the linked list stored at index `hash(entry.key)`.
If we have 128 entries to store, and the maximum size of the linked list allowed is 3, and the hashing function is $h(k) = k \bmod m$, then the best value of m would be 128/3 rounded above, i.e. 43. We should ensure that the number is also a prime, and not close to a power of 2.
The worst case for insertion and deletion is therefore $O(1)$. However, for search, it can be linear- this can happen if all the keys hash to the same slot, and depends on the hash function. The average case depends on how the hashing function distributes the keys over the buckets.
Now, we make the simple uniform hashing assumption (SUHA)- the function distributes the keys evenly over the buckets; every item has an equal chance of getting put in a bucket, irrespective of what is already in the chain. Moreover, we also assume that the hash function can be computed in constant time. Then, we define the load factor for the table to be $\alpha = n/m$, where n is the number of keys in the hash table, and m is the number of slots available. In the example

above, $\alpha = 128/43 \approx 3$. If this assumption cannot be made, then the hashing function isn't efficient.
With SUHA, we re-analyse the search algorithm. If the search is unsuccessful, we need to scan a particular linked list completely. The average length of the chain is α, and computing the hash is constant. So, the running time of the function is $O(1+\alpha)$. In particular, if α is a constant, the function runs in constant time.
Now, we assume that the search is successful. The probability that a chain is searched is proportional to the number of elements within it, that is, not every chain is equally likely to be searched. After selecting a chain, we need to scan a chain with all the elements that was added after it, along with the element itself. Now, the probability we want to find is:

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right],$$

where E denotes the expectation operator, and the value i refers to the i-th element inserted to the table. Moreover, X_{ij} is a random variable (value 1 or 0) storing whether the hash of the i-th element equals the has of the j-th element. With SHUA, we know that $E[X_{ij}] = 1/m$, so on average,

$$\begin{aligned}
E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] &= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right)\\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)\\
&= \frac{1}{n}\sum_{i=1}^{n}1+\frac{1}{mn}\sum_{i=1}^{n}\sum_{j=1}^{n}1\\
&= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)\\
&= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right)\\
&= 1+\frac{n}{m}-\frac{n+1}{2m}\\
&= 1+\frac{2n-n-1}{2m}\\
&= 1+\frac{n}{2m}-\frac{1}{2m}\\
&= 1+\frac{1}{2}\alpha-\frac{1}{2n}\alpha.
\end{aligned}$$

So, the running time of the function is $O(1+\alpha)$, like in the unsuccessful search. So, on average, searching takes constant time using hash tables with chaining if $O(n) = O(m)$.

Now, we look at a different method of collision resolution, called probing (or open addressing). Here, if a collision occurs, we try to add the value to a different cell. So, we avoid using linked lists (i.e. space).

To find the next cell to search, we have a function that takes two values- the key, and the time we're probing (starting from 0). This is an extension to the normal hash function. In this case, we require all the locations within the hash table to be reached. This method is a better choice if memory is limited.
The insertion function in this case becomes:

```
int insert<K, V>(Map<K, V> map, MapEntry<K, V> entry):
    int i = 0
    while (i < map.table.length):
        int j = hash(k, i)
        if (map.table[i] == null):
            map.table[i] = entry
            return j
        else:
            i++
    throw Error("Overflow!")
```

Here, we return the index where the element is inserted. The function has runtime $O(m)$, which we expect to be a constant.
We consider how this function works when adding entries into the table

null	113:12	null	83:8	null	109:2	null	71:9

with the hashing function $hash(k, i) = (k + i) \bmod 8$. First, we add `(65, 6)`.

null	113:12	65:6	83:8	null	109:2	null	71:9

During the first iteration, we get 1. But, index 1 is occupied, so we run the hash function again. This gives us 2, and since index 2 is empty, we add it there. Next, we add `(57, 9)`.

null	113:12	65:6	83:8	57:9	109:2	null	71:9

Here, the hashing function gives us 1 in the first iteration. So, we need to iterate two times more to get a free index- 4.
Next, we consider the search function:

```
V search(Map<K, V> map, K key):
    int i = 0
    while (map.table[i] == null || i < m):
        int j = hash(key, i)
        if (map.table[j].key == key):
            return map.table[j].value
        i++
    return null
```

This operates in a similar manner to the insert operation.
The delete function is more complicated since we need to ensure that the probing function is not broken. We could make it such that deleted slots were ignored by the insert function. The search function doesn't get changed since the deleted value will be empty. However, this means that the running time of search doesn't depend on the load factor.
Now, we look at probing algorithms. The probing algorithm $h(k, i) = (h'(k) + i) \bmod m$, with h' the normal hashing function, is called linear probing, and is

not the best approach. It is however very easy to implement, but the average search time can increase if there is a build up of the occupied slots- this is primary clustering.
Another possibility is to have quadratic probing. Here, the hash function becomes $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, where c_1, c_2 are constants. This is better than linear probing, and more efficient if c_1, c_2, m are constrained. However, if two keys have the same probing value, their probing sequence will be the same- this is called secondary clustering, but is a milder form of clustering than primary clustering.
We can also use double hashing. Here, the hashing function is $h(k,i) = (h_1(k) + ih_2(k)) \bmod m$, where h_1 and h_2 are normal hashing functions. We need $h_2(k)$ to be relatively prime to the size of the table m to ensure that all the indices are returned by the function. We can, for example, let m be a power of 2, and let h_2 always be odd, or let m be a prime, and let h_2 be a positive integer less than m. This is one of the best methods for open addressing.
On average, an unsuccessful search (or insertion), we need to probe

$$1 + \alpha + \alpha^2 + \alpha^3 + \cdots = \frac{1}{1-\alpha}$$

times. This is because the probability that the first probe is occupied is α; the first two probes are occupied is α^2, and so on. We also need to account for the first probe. So, if α is a constant, these operations have runtime $O(1)$. On the other hand, if the search is successful, then the number of probes we need is

$$\frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha} \right).$$

Here as well, if α is a constant, the operation has runtime $O(1)$.

Sometimes, we know the once the keys are stored within the table, they will not get changed. Then, we can use perfect hashing that ensures there will be no collisions. So, both successful and unsuccessful search run in constant time. There are two levels of hashing here, both of which are universal:

- The first hashing hashes like with collision resolution by chaining, which gives us a bucket to attach the element.
- The second hashing takes the location where we would add the element into the linked list and converts it into a value that avoids collisions.

2.5 Advanced Design Techniques

We have looked at 3 algorithm design techniques: incremental algorithms (e.g. insertion sort), divide and conquer (e.g. merge sort) and randomisation (e.g. randomised quick sort). We now study a new technique by considering the knapsack problem.

The knapsack problem asks the following question: if we have n items of different weights in different numbers, which elements do we choose to maximise the amount of items for a fixed weight. The items need not equal the weight. This is a combinatorial optimisation problem. We can look at this problem under 2 circumstances: the "0-1 knapsack problem" (where we cannot have a fraction of an item), and the "fractional knapsack problem" (where we can take a fraction of an item).

0-1 Knapsack problem

We now consider the 0-1 knapsack problem abstractly. We are given two lists `A` and `B` of some fixed size `n`, and we want to take a subset of `A` that gives the highest value given that the corresponding indices in `B` are below a fixed value. For example, if the constraint is 16, and we have the lists: `A = [5, 3, 4, 2]` and `B = [12, 5, 3, 1]`, then we take the subset `[5, 4, 2]`- there is no better choice of a subset of `A`. In particular, we could have chosen `[3, 4, 2]`, but the sum of this list is smaller than the one given above.

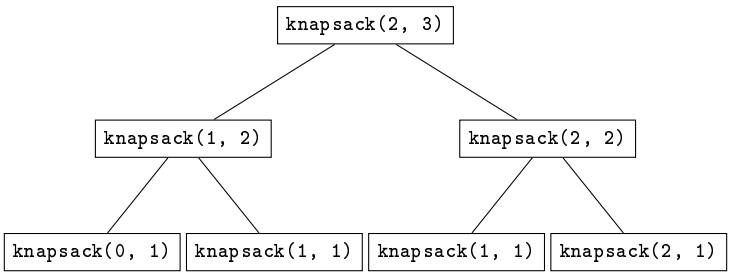
We can go through all the possible combinations and select the one which satisfies the two constraints. This is a brute force approach. Since an array of size `n` has 2^n possible combinations, this algorithm would have runtime $O(2^n)$- this is not an efficient approach.

Instead, we can take a divide and conquer approach. We consider all the subsets, but loop around an element to decide whether to add it to the subset by checking if the sum of list `B` is still below the constraint, and we are at an advantage without adding it- we take the maximum of the two possibilities. The following is the pseudocode for this approach:

```
int knapsack(int constraint, Array<int> A, Array<int> B, int i):
    if (i < 1 || constraint = 0):
        return 0
    // if B[n] is higher than the constraint, can't add it to the
    subset
    if (B[n] > constraint):
        return knapsack(constraint, A, B, i-1)
    else:
        // choose to include this elt or not depending on the
    values -> want the maximum
        int included = A[n] + knapsack(constraint-B[n], A, B, i-1)
        int excluded = knapsack(constraint, A, B, i-1)
        return maximum(included, excluded)
```

The value returned here is the maximum possible sum of the subset of `A` which satisfies the constraint.

We consider the recursion tree of the code above for the case `constraint = 2, A = [5, 8, 7], B = [1, 1, 1]`, starting with `i = 3` (note: in the diagram, `knapsack(2, 3)` means `knapsack(2, A, B, 3)`):



We can see that the algorithm is not efficient since `knapsack(1, 1)` runs twice. In fact, the running time of this function is still $O(2^n)$. Nonetheless, the concept has a lot of potential under a different approach- dynamic programming.

Dynamic programming

Here, the word 'programming' refers to storing values in a table. This makes the algorithm run on a better complexity, but requires some space (a 2×2 array). There are 4 steps to any dynamic programming algorithm:

1. We first need to understand the optimal solution.
2. We need to then recursively define the optimal solution.
3. We then adapt the optimal solution with the help of a 'table'. This allows us to find the optimal value.
4. We can then find out which elements are part of the optimal solution, if required.

In the case of the 0-1 knapsack problem, we already have recursive version of the optimal solution. Now, we construct the table- this is a 2D array of numbers. In this case, the entry `table[i][j]` stores the maximum total value of any subset of items $\{1, \cdots, i\}$ of combined weight at most `j`. So, the entry `table[n][W]` contains the optimal solution, where n is the length of the 2 arrays, and W is the constraint. In terms of the recursive algorithm, the base case means that `table[0][j] = 0` for all `j`. Also, `table[i][j]` returns the lowest possible integer if `j < 0` since negative indexing is not possible. The recursive step is

`table[i][j] = max(table[i-1][j], A[i] + table[i-1][j-B[i]])`.

Here, `i` ranges from `1` to `n`, while `j` ranges from `0` to `constraint`.
Next, we construct a bottom-up implementation of the recursive algorithm using a table. For example, assume that `constraint=7`, `A=[5, 2, 1, 4]` and `B=[2, 3, 4, 1]`. First, we initialise the table:

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								

Then, we add the base case- the top row is initialised with value 0.

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1								
2								
3								
4								

We now compute the values in row 1.

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2								
3								
4								

This is about including the element `5` into the list, and requires the constraint to be `2` or more. This is reflected in the table above. In particular, when computing `table[1][0]`, we take the maximum of `table[0][0]` and `5 + table[0][-2]`- since `-2` isn't a valid index, the maximum is `table[0][0] = 0`. On the other hand, if we want to compute `table[1][2]`, we take the maximum of `table[0][2]` and `5 + table[0][0]`, so we have `5`. Now, we compute the values in row 2:

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3								
4								

Since `B[1] = 3`, the values of `table[2][0]` and `table[2][1]` remain 0. There is no competition for `table[2][2]` since this is the maximum of `table[1][2] = 5` and `2 + table[2][-1]`. On the other hand, when we compute `table[2][3]`, we need to take the maximum of `table[1][3] = 5` and `2 + table[2][0] = 2`- the value remains 5. It is only when the weight is high enough for us to accommodate them both that we incorporate `A[2]`. In particular, when we compute `table[2][5]`, we take the maximum of `table[1][5] = 5` and `2 + table[1][2] = 7`. Next, we compute row 3:

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4								

We have `B[2] = 4`, so the values `table[3][0], table[3][1], table[3][2]` and `table[3][3]` get copied from the row above. When we find `table[3][4]`, we take the maximum of `table[2][4] = 5` and `1 + table[2][0] = 1`, so

there still is no change. In fact, there is no change for all the cells in this row. Now, we compute the final row:

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4	5	9	9	9	11	11

Since `B[3] = 1`, we can make `table[4][1] = 4`. But, `table[4][2]` is the maximum of `table[3][2] = 5` and `4 + table[3][1] = 4`, so that remains 5. However, `table[4][4] = 9` because it is the maximum of `table[3][4] = 5` and `4 + table[3][3] = 9`. This continues until we compute `table[4][6]`, which takes the maximum of `table[3][6] = 7` and `4 + table[3][5] = 11`. Therefore, we have found that the maximum sum of the subset of `A` provided the corresponding indices sum to a value below 7 is `11`.
The pseudocode for the dynamic programming algorithm of the knapsack is:

```
int knapsack(int constraint, Array<int> A, Array<int> B, int n):
    // table with length of arrays n+1, with each array of length
    constraint+1
    Array<Array<int>> table = []
    // initialise the table
    for (int j = 0; j <= constraint; j++):
        table[0][j] = 0
    // pick the maximum
    for (int i = 1; i <= n; i++):
        for (int j = 0; j <= constraint; j++):
            // if we can add this element, then take the maximum
            if (B[i] <= j):
                table[i][j] = max(table[i-1][j], A[i] + table[i
    -1][j-B[i]])
            else:
                table[i][j] = table[i-1][j]
    // the bottom right element is the optimal solution
    return table[n][constraint]
```

This algorithm has running time $O(nW)$, where W is the constraint.
We now use the table to find the indices in the optimal solution- this is step 4. The pseudocode for the procedure is:

```
List<int> indicesInOptimal(Array<Array<int>> table, Array<int> A,
    Array<int> B, int constraint):
    int i = A.length
    int k = constraint
    List<int> indices = []
    while (i > 0 && k > 0):
        if (table[i][k] != table[i-1][k]):
            indices.add(i)
            k = k - B[i]
        i--
    return indices
```

Here, we traverse the rows of the table from the bottom, adding the index if the value is greater than the one in the row above. It has (worst case) runtime $O(n)$, where n is the length of the two arrays.
We show which indices are selected using the table above. We start with

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4	5	9	9	9	11	11

Since `7` and `11` aren't equal, we add `4` to the `list`. The value of `i` decreases by 1, while `k` decreases by `B[4] = 1`.

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4	5	9	9	9	11	11

Here, `table[3][6] = table[2][6]`, so `3` doesn't get added to the list. We just decrement the value of `i`.

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4	5	9	9	9	11	11

Here, `table[1][6]` and `table[2][6]` do not match, so we add `1` to the indices list. Both `i` and `k` get decremented.

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	5	5	5	5	5	5
2	0	0	5	5	5	7	7	7
3	0	0	5	5	5	7	7	7
4	0	4	5	9	9	9	11	11

Since `5` and `0` are different, `0` also gets added to the indices list. Now, `i` is 0, so the while loop terminates. So, the optimal solution contains elements in indices `1`, `2` and `4`.

Memoisation

Instead of the dynamic programming approach, we could have also used memoisation. Like with dynamic programming, the recursion values are stored for a subproblem so that they are only computed once- this is a memoised recursion; it remembers what has already been computed.
For the case of the knapsack problem, the following is the memoised version of the problem:

```
int knapsack(Array<Array<int>> table, Array<int> A, Array<int> B,
    int n, int constraint):
    if (n < 1 || constraint == 0):
```

```
3           return 0
4       // memoised
5       if (table[constraint][n] != -1):
6           return table[constraint][n]
7       if B[n] > constraint:
8           table[constant][n] = knapsack(table, A, B, n-1, constraint
        )
9       else:
10          int included = A[n] + knapsack(table, A, B, n-1,
        constraint-B[n])
11          int excluded = knapsack(table, A, B, n-1, constraint)
12          // saving the value into the table so that it doesn't need
         to be computed again
13          table[constraint][n] = max(included, excluded)
14      return table[constraint][n]
```

Originally, we initialised the table with `-1` in all rows and columns. In the recursive algorithm, we returned the value as soon as we knew what it was. Here, we first add it to the table, and then return. This algorithm also has runtime $O(nW)$, where n is the length of `A` and `B`, and W is the constraint. However, since the algorithm is recursive, we need to maintain the stack. So, it might be less preferable than the dynamic programming version.

Fractional knapsack problem

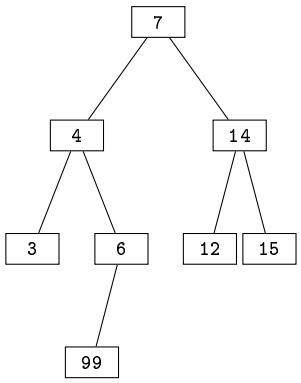
This is very similar to the 0-1 knapsack problem, but we allow for fractions of an element to be taken. This makes the problem much simpler. The solution can be found in 2 steps:

- We create an array `ratio` containing the values of `A[i]/B[i]`, and sort the arrays `A` and `B` with respect to the `ratio`.
- We then take the maximum amount possible, including fractions to ensure that the constraint is met.

This is a greedy approach.

Greedy algorithm

Greedy algorithms are common in optimisation problems. Here, every choice we make is locally optimal. It is much faster than dynamic programming since we search for a globally optimal solution there.
In the case of the fractional knapsack, if we have `constraint = 7, A = [5, 2, 1, 4], B = [2, 3, 4, 1]`, then we get `ratio = [2.5, 0.67, 0.25, 4]`. We can sort `A` and `B` with respect to the ratio, and get `A = [4, 5, 2, 1], B = [1, 2, 3, 4]`. Then, we take the values from `A` until we get the corresponding indices in `B` crossing 7. So, we get `[4, 5, 2, 0.25]`. The optimal solution is therefore `11.25`. This algorithm is much more efficient than the dynamic programming approach.
However, it can be difficult to say whether the greedy approach will give the best solution. For example, consider the problem of finding a path from the root to a leaf in the following binary tree with the maximum sum:



If we always took the locally maximum value, we would get `7, 14, 15`. However, the globally maximum solution is `7, 4, 6, 99`.