Data Structures

PIC 10B, UCLA

Data Structures

There are several means of storing data: data can be stored contiguously such as with a **std::vector** or **std::deque**; as a series of “nodes” in a tree with an associative structure, such as a binary search tree, serving a basis for **std::set**s and **std::map**s; or as a series of nodes without an associative structure such as a **std::list**. It is also possible to combine these ideas into objects such as hash tables. Here, we will study and implement the fundamental data structures and examine some special adapter classes that build upon the fundamentals.

Containers of the Standard Library

We will briefly look at the different container clases available in the C++ Standard Library. As will become quite evident, there are many varieties of structures beyond simply a **std::vector**s or **std::set**s.

Containers of the Standard Library

Some things to watch for:

- Most of the containers have iterators that can go in either direction, admit range-for loops, etc.
- Most containers have an emplace functionality: being able to construct an object in place by passing parameters for its construction, usually done with a member function called **emplace** or **emplace_back** or **emplace_front**.
- Most containers have an **empty** function to query if the container is empty.
- Most containers use **push** or **push_front** or **push_back** to insert elements at the front/back and use **pop** or **pop_front** or **pop_back** to remove elements at the front/back.
- Most containers can be initialized from a **std::initializer_list**, i.e., listing the initial values it stores in **{ }**'s or they can start empty during default construction.

The examples given are not complete illustrations of how to use the containers; they are meant only as an overview of what variety of structures are out there.

List Structures

Structures based on lists can store lists of values. They can grow at one or both ends but their data *are not stored contiguously*. In particular, they do not support random access to their elements.

Given a position within a list, inserting an item has an $O(1)$ cost (unlike a vector with $O(n)$ cost), but this is seldom an advantage because getting to an arbitrary position in a list has an $O(n)$ cost.

Historically lists were useful when it was difficult to obtain a large amount of contiguous memory. On modern machines, this is seldom a problem.

std::list I

std::list is found in the **<list>** header.

In the interface, lists behave somewhat like **std::vector**s or **std::deque**s.

std::list II

Items can be added at the front with **push_front** and at the back with **push_back**; items can be removed from the front and back with **pop_front** and **pop_back.**

Neither container allows random access (no subscripting, no adding/subtracting integers from an iterator).

For a list, we can use both **emplace_front** and **emplace_back.**

std::list III

```
std::list<int> L{1,2,3};

auto itr = std::begin(L); // itr is of type std::list<int>::iterator

L[2]; // ERROR: no subscripts
itr - 1; // ERROR: no pointer arithmetic
```

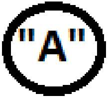
std::list IV

```
std::list<std::string> List {"A"}; // starts with single element "A"

List.emplace_front(3, 'C'); // adds object given constructor params
List.push_back("BB");
List.emplace_front(); // adds empty string: no parameters given
List.pop_front();           List.pop_back();

auto itr = List.end();
- -itr; // now points to last item: "A"
List.insert(itr,"HI"); // places "HI" before last element

for( const auto& s : List) {
   std::cout <<s <<" ";
}
```

```
CCC HI A
```

List Code Illustration

```
std::list<std::string> List{"A"};
```



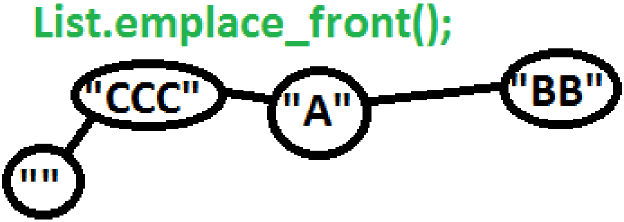
List Code Illustration

```
List.emplace_front(3, 'C');
```

List Code Illustration

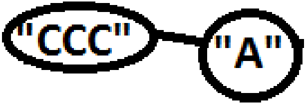
```
List.push_back("BB");
```

List Code Illustration



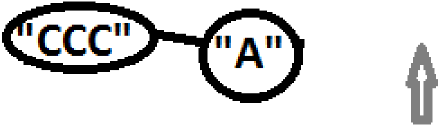
List Code Illustration

List.pop_front(); List.pop_back();



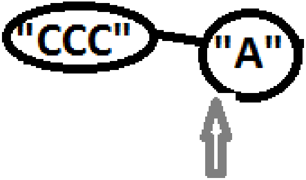
List Code Illustration

```
auto itr = List.end();
```



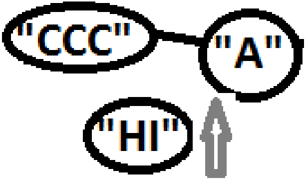
List Code Illustration

```
--itr;
```

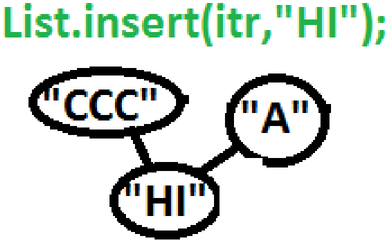


List Code Illustration

```
List.insert(itr,"HI");
```



List Code Illustration



Random Access Containers

Containers that support random access (subscripting, say) are often implemented as a contiguous block of memory (not always, though).

They offer a large benefit in that the *i*th element can be accessed with a cost of $O(1)$ because the address can be found immediately.

They have a disadvantage in that inserting an element at an arbitrary position has an $O(n)$ cost because as many as all items may need to be moved.

std::array I

An **std::array** is a templated class that operates as a C-style array (like an **std::vector** but with a constant size). It is found in the **<array>** header.

The template requires a type to store and a size. The size cannot be changed.

Like most containers, it has iterators and even member functions to return iterators (recall a static array does not have member functions).

Unlike **std::vector**, an **std::array** has stack-based data making for faster element access - at least for small to moderate amount of data. Once there are lots of data, it may be better to use a **std::vector** to take advantage of move semantics.

std::array II

```
// array of size 4 doubles: 1, 2, then two 0-values
std::array<double, 4> doubleArray = { 1.3, 2.2 } ;

doubleArray[3] = 7; // has subscripts

for (auto itr = doubleArray.rbegin(), end = doubleArray.rend(); itr != end;
   ++itr) {
   std::cout <<*itr <<" "; // has iterators
}
```

```
7 0 2.2 1.3
```

std::vector and std::string I

A **std::vector** is a templated class that stores objects of a given type; a **std::string** behaves nearly identically to a **std::vector** but its elements are **char**s.

Both data structures have similar types of constructors, have subscripts, have **push_back** and **pop_back**, etc.

std::string has more non-iterator related functions like **std::string::find**, whereas for **std::vector** we'd need to use the generic algorithm **std::find**, etc.

std::vector and std::string II

```
std::string s(3, 'a'); // s == "aaa"
std::vector<int> v(3, 8); // v == {8, 8, 8}

s[1] = 'b'; // s == "aba"
v[2] = 10; // v == {8, 8, 10 }

s.pop_back(); // s == "ab"
s.push_back('C'); // s == "abC"
```

std::deque I

A **std::deque** (short for **d**ouble-**e**nded **que**ue), included in the **<deque>** header, is a vector-like structure that can grow/shrink both at its front and back in $O(1)$ time.

std::deque II

```
std::deque<int> dint(4); // will store 4 0-initialized int values

dint[1] = 1; // set second int to 1

dint.pop_back(); // pop off last element
dint.pop_front(); // pop off first element

dint.push_front(100); // append 100 to beginning

for (int i : dint ) { // print all the ints
   std::cout <<i <<" ";
}
```

```
100 1 0
```

Deque Code Illustration

```
std::deque<int> dint(4);
```

0 0 0 0

Deque Code Illustration

```
dint[1] = 1;
```

0 1 0 0

Deque Code Illustration

```
dint.pop_back();
```

0 1 0

Deque Code Illustration

```
dint.pop_front();
```

➡

1 0

Deque Code Illustration

```
dint.push_front(100);
```

<u>100</u> <u>1</u> <u>0</u>

Associative Containers

The associative containers store their data in a "sorted order".

Elements can be inserted with a cost of $O(\log n)$ and an element can be removed with a cost of $O(\log n)$.

They do not support random access, but iterators can be used to traverse, search, etc.

std::set I

A **std::set** is a templated data structure that stores data in a sorted order (by default based upon **operator<**).

The underlying data structure is not specified by the Standard but it must support operations such as insertion, lookup, etc., with an $O(\log n)$ cost no matter how the structure is used.

Random access and iterator arithmetic are not supported: we can only increment/decrement iterators.

There can be at most one occurrence of each element.

std::set II

```
std::set<int> set; // default
for (size_t i = 0; i < 10; ++i) {
   set.insert( std::rand() % 5 ); // add 0, 1, 2, 3, or 4
} // maybe set == { 0, 1, 2, 4 } (no 3's came up)

for (auto it = set.begin(), end = set.end(); it != end; ++it) {
   if ( *it == 2) {
      set.erase(it); break;
   } // erase 2, so set could be { 0, 1, 4 }
}
```

std::set III

```
for ( auto it = set.rbegin(); it != set.rend(); ++it) { // print in reverse order
   std::cout <<*it <<" ";
}

auto it = set.find(8); // will point to end if not found
std::cout <<( (it != set.end() ) ? "found": "not found");
```

```
4 1 0 not found
```

std::map I

A **std::map**, found in the **<map>** header, is a templated class that stores key-value pairs. Each key is given a value and the keys are stored in order based on **operator<** by default.

As with **std::set**, random access and iterator arithmetic are not supported. There can be at most one occurrence of each element.

Values are accessed (or created and *value initialized*!) from their key with **operator[]**. To prevent creating an item, **at** can be used instead of **operator[]** throwing an exception if a key is not present.

Iterators manage **std::pairs** and range-for loops are done over **std::pairs**.

std::map II

```
// default construct
std::map<std::string, double> employeeSalaries;

// create entries via subscripting
employeeSalaries["Cindy"]; // creates "Cindy" key with value 0!
employeeSalaries["Bob"] = 98000; // by key, "Bob"< "Cindy"
employeeSalaries["Alice"] = 200000;
```

std::map III

```
// find iterator to given key or returns end iterator
auto it = employeeSalaries.find("Bob");
if (it != employeeSalaries.end() ) { // if Bob found, change salary
    it->second = 150000; // it manages a pair
}
// or we could have just reassigned:
// employeeSalaries["Bob"] = 15000;

employeeSalaries["Cindy"] = 60000; // gives "Cindy" key a value of 60000

for ( const auto& pair : employeeSalaries ) {
    std::cout <<pair.first <<"-"<< pair.second << '\n';
}
```

std::map IV

```
Alice-200000
Bob-150000
Cindy-60000
```

Map Code Illustration

```
std::map<std::string,double> employeeSalaries;
```

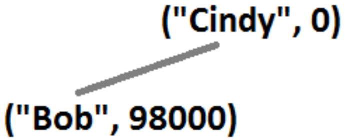
Map Code Illustration

```
employeeSalaries["Cindy"];
```

("Cindy", 0)

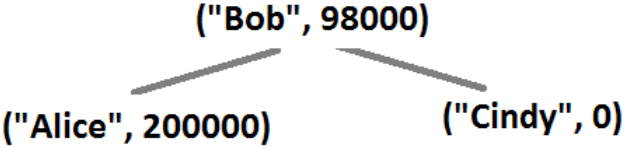
Map Code Illustration

```
employeeSalaries["Bob"] = 98000;
```



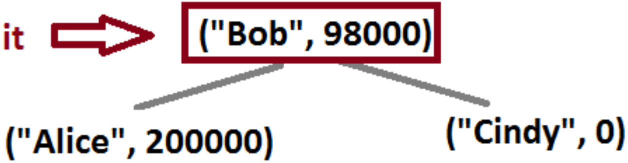
Map Code Illustration

```
employeeSalaries["Alice"] = 200000;
```

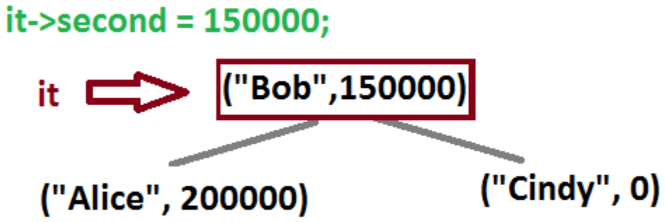


Map Code Illustration

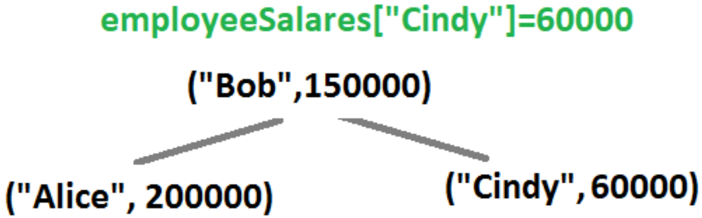
```
auto it = employeeSalaries.find("Bob");
```



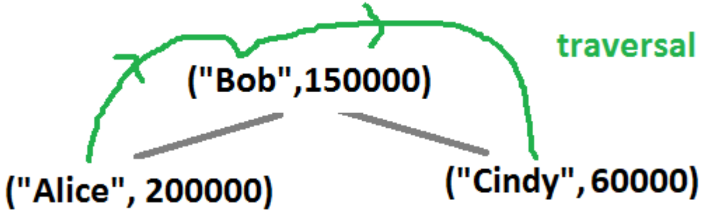
Map Code Illustration



Map Code Illustration



Map Code Illustration



std::multiset and std::multimap I

A **std::multiset** found in the **<multiset>** header functions the same as **std::set** but can store repeated elements.

A **std::multimap** found in the **<multimap>** header functions the same as **std::map** but multiple instances of a key for **std::multimap** are possible. The Values are not sorted.

Pairs of values can be directly inserted into a **std::multimap** with **insert**.

Using **count** we can obtain how many times a given key appears in a **std::multimap** or a given value occurs in a given **std::multiset**. It can also be used in **std::map** and **std::set** but the value will only ever be 0 or 1.

std::multiset and std::multimap II

```
std::multimap<int, int> mm;

mm.insert({ 10, 1 }); // can directly insert pairs for map/multimap, etc.
mm.insert({ 5, 3 });
mm.insert({ 100, 11 });
mm.insert({ 100, 4 }); // have more than one key of 100

std::cout << mm.count(100) <<" times" <<'\n'; // count instances

for (auto it = mm.begin(), end = mm.end(); it != end; ++it) {
   std::cout << it->first << " " << it->second << '\n';
}
```

std::multiset and std::multimap III

```
2 times
5 3
10 1
100 11
100 4
```

Hash Maps

Some containers are implemented as **hash maps**. The data are not stored contiguously nor are the data sorted.

The hash maps require a hashing function to process the data, a means of assigning an index to each value and placing it.

Hash tables have an advantage of having $O(1)$ lookup, insertion, and removal!

Unordered Sets and Maps I

There are also **std::unordered_set** and **std::unordered_multiset** in **<unordered_set>**, and **std::unordered_map** and **std::unordered_multimap** in **<unordered_map>**.

They behave similar to their ordered counterparts in syntax but the data are not stored in sorted order! Instead, the compiler uses a *hash function* (to be discussed) to place the elements.

These data structures are useful if there is no **operator<** that makes sense for the data.

These structures also provide $O(1)$ lookup to any given element and $O(1)$ insertion. This is better than any other data structure.

Unordered Sets and Maps II

```
// hashes ints, can have value appearing multiple times
std::unordered_multiset<int> ums;

for (size_t i = 0; i < 10; ++i) {
   ums.insert(std::rand() % 5);
}

for (auto it = ums.begin(); it != ums.end(); ++it) {
   std::cout << *it << " "; // data not sorted
}
```

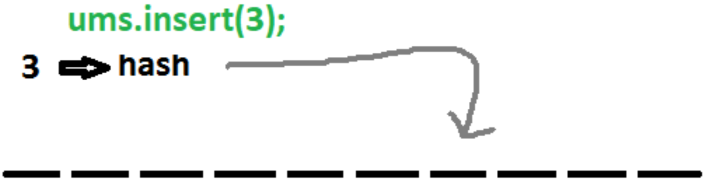
```
4 0 0 0 1 1 3 3 2 2
```

Unordered Multiset Illustration

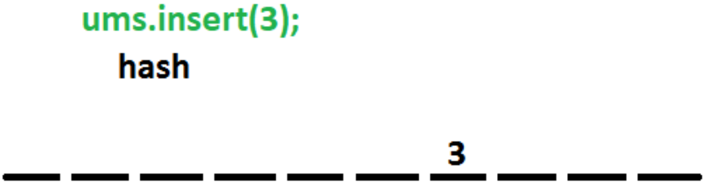
```
std::unordered_multiset<int> ums;
```

hash

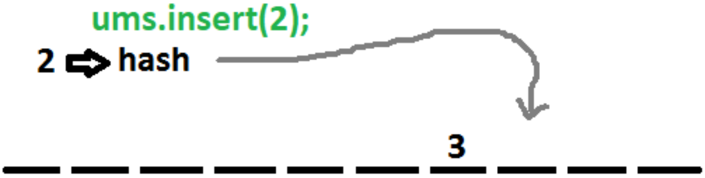
Unordered Multiset Illustration



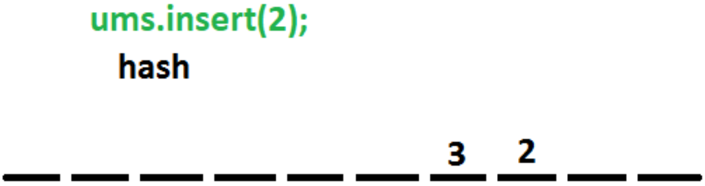
Unordered Multiset Illustration



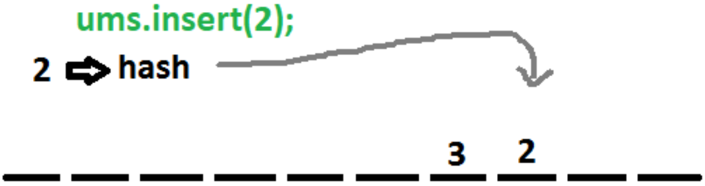
Unordered Multiset Illustration



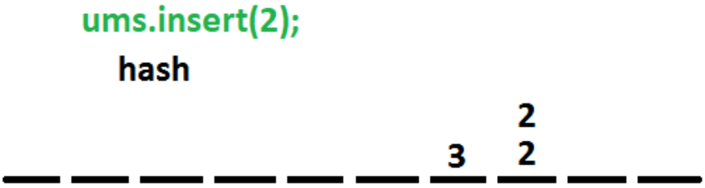
Unordered Multiset Illustration



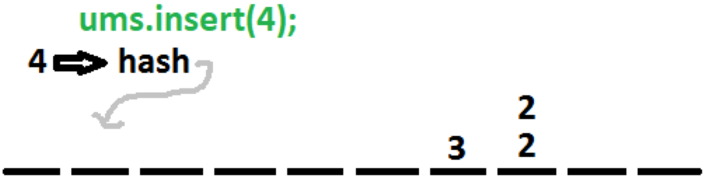
Unordered Multiset Illustration



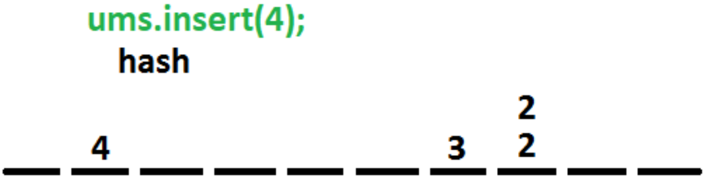
Unordered Multiset Illustration



Unordered Multiset Illustration



Unordered Multiset Illustration



Other Containers

Some containers are built upon other more fundamental containers, so called container adapters.

Other containers are just different altogether and don't quite fit into the previous categories.

std::priority_queue I

A **std::priority_queue**, defined in the **<queue>** header, is a templated class that provides fast access to the element of "highest value".

By default this "highest value" is based on **operator<**.
Values are added in with a **push** and the highest value is removed with a **pop** or viewed with a **top**.

There are no iterators for this structure, no subscripting, etc.

Conceptually it is built upon a **maximum heap** (to be discussed).

std::priority_queue II

```
std::priority_queue<int> pque;

pque.push(10); // add new elements
pque.push(20);
pque.push(6);

std::cout <<pque.top() <<" "; // access highest priority item: 20

pque.pop(); // remove the highest item: 20 so 10 is new highest

std::cout <<pque.top(); // access highest item: 10
```

```
20 10
```

Priority Queue Illustration

```
std::priority_queue<int> pque;
```

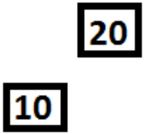
Priority Queue Illustration

```
pque.push(10);
```

$\boxed{10}$

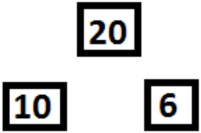
Priority Queue Illustration

```
pque.push(20);
```



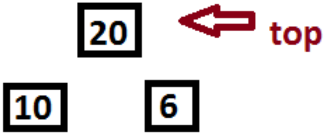
Priority Queue Illustration

```
pque.push(6);
```



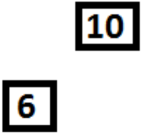
Priority Queue Illustration

```
std::cout << pque.top();
```

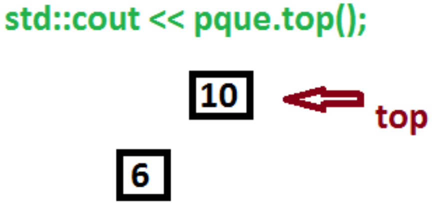


Priority Queue Illustration

```
pque.pop();
```



Priority Queue Illustration



std::stack I

A **std::stack**, defined in the **<stack>** header, is a templated class that operates as a LIFO (last in, first out) data structure... as the name suggests.

Many containers have an **empty** function that tells us whether the structure is empty. This can be useful to avoid doing operations on a data structure that would be invalid if it were empty to begin with.

Items are added with a **push**, viewed with a **top**, and removed with a **pop**.

There are no iterators for this structure, no subscripting, etc.

std::stack II

```
std::stack<std::string> stringStack;

stringStack.push("abc"); // add to top of stack
stringStack.push("def"); // added on top of "abc"

if ( ! stringStack.empty() ) { // if not empty
   stringStack.pop(); // pop off top: pops off "def"
}

if (! stringStack.empty() ) { // if not empty
   std::cout <<stringStack.top(); // show the top
}
```

```
abc
```

std::queue I

A **std::queue**, defined in the **<queue>** header, is a templated class that operates by as a FIFO (first in, first out) data structure. Items are removed from the front and added to the back; we can only access the front and back elements.

Items are added (to the back) with **push** and removed from the front with **pop**; the front and back elements are accessed with **front** and **back**.

There are no iterators for this structure, no subscripting, etc.

std::queue II

```
std::queue<int> qint;
qint.push(1); // add items to the queue
qint.push(2);
qint.push(3); // so 1 is first, 2 is second, 3 is third

std::cout <<qint.front(); // see the front element

qint.pop(); // can only remove the front element

std::cout <<qint.front();

// modify the front and back
qint.front() = 8;
qint.back() = 9;
std::cout <<qint.front() <<qint.back();
```

```
1289
```

std::pair I

A **std::pair** is a templated class that stores two public members.

It is defined in the **<utility>** header. The elements are called **first** and **second**.

The **std::make_pair** function returns a **std::pair** based on deducing the types of its input arguments (or they can be explicitly listed).

std::pair II

```
// value initialize first default init second, first == 0, second == ""
std::pair<int, std::string> pair;

pair.first = 7; // access/modify first
pair.second = "lucky number"; // access/modify second
std::cout <<pair.second <<" "<< pair.first <<'\n';

pair = std::make_pair(101, " dalmatians");
std::cout <<pair.first <<" "<<pair.second <<'\n';

/* Or:
pair = std::make_pair<int, std::string>(101, " dalmatians"); */
```

```
lucky number 7
101 dalmatians
```

std::initializer_list I

A **std::initializer_list** is a templated class found in the **<utility>** header. It stores a braced list of values of the same type and can be useful for initializing a class.

```
// std::vector<double> constructed with std::initializer_list<double>
std::vector<double> vec = { 2.2, 9.8, 37, -11.14 };
```

Functionalities are mostly limited to working with iterators, range-for loops, and a **size** function.

std::initializer_list II

```
struct TEN_INTS { // this array is constructed with an initializer list
    static constexpr size_t CAPACITY = 10;
    int point[CAPACITY]; // static array of 10 ints

    TEN_INTS ( const std::initializer_list<int>& list ) {
        size_t pos = 0; // position in the array

        // loop over list: bad things happen if list size exceeds 10!
        for (int i : list) {
            point[pos++] = i; // place value in spot
        }
    }
};
```

std::initializer_list III

With the **TEN_INTS** class above, we could write:

```
TEN_INTS myInts { 1, 2, 3, 4, 5 };
```

and have the class store: { 1, 2, 3, 4, 5, ?, ?, ?, ?, ? } (only the first five values are initialized).

Logic of Data Structures

Having seen many different containers, we will examine some of the relevant computer science ideas behind many of these containers.

Linked Lists

Recall that a **std::vector** or C-style array store their data contiguously. This makes lookup and entry modifications exceedingly fast: the address can easily be computed.

There are some small drawbacks, though: an array has a fixed size and a vector, when its size reaches the capacity, must allocate another contiguous block of memory, transfer all the entries over, then add more elements.

In practice, these are hardly problematic on modern machines. There was a point, though, when finding a large block of contiguous memory was difficult and for this reason along with various theoretical ideas (which fail in practice), other means of storing data were considered such as a **linked list**.

Linked Lists

A linked list is a data structure where data are stored in “nodes”: the nodes are not contiguous, but they know where their left/right neighbours live in memory.

Linked List Visual

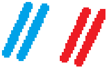
Let’s imagine that we have a linked list structure and we wish to add a series of **int** values for it to store. We might add, for example, the numbers to store: 5, 8, 0, 3.

The list starts empty and the first (blue) and last (red) nodes are null (// indicates a null value, **nullptr**). The first and last nodes are always tracked.

As numbers are added, a new node is created which tracks the node before it (grey) and the next node (green).

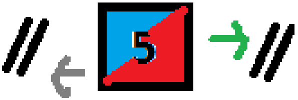
Linked List Visual

The list starts empty: the values 5, 8, 0 and 3 are added in that order.



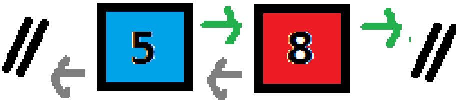
Linked List Visual

The list starts empty: the values 5, 8, 0 and 3 are added in that order.



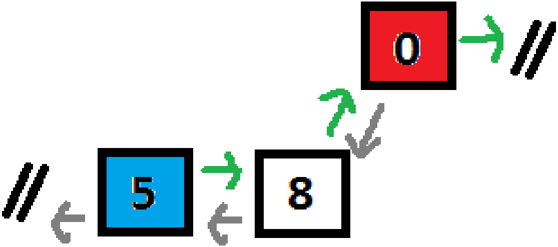
Linked List Visual

The list starts empty: the values 5, 8, 0 and 3 are added in that order.



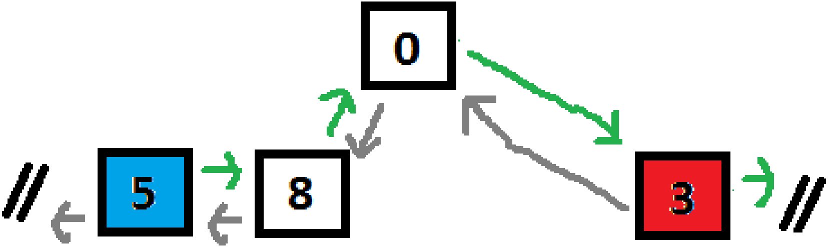
Linked List Visual

The list starts empty: the values 5, 8, 0 and 3 are added in that order.



Linked List Visual

The list starts empty: the values 5, 8, 0 and 3 are added in that order.



Linked Lists

Using a linked list requires 3 classes that work together: the **linked list** itself, a **node** class to describe the nodes that store the data and point to other nodes, and an **iterator** class to safely traverse the list and assist in insertions and deletions.

We begin by considering the functionalities of these classes and then define their interfaces. Each class is extremely basic, storing nothing but the essentials.

Linked Lists

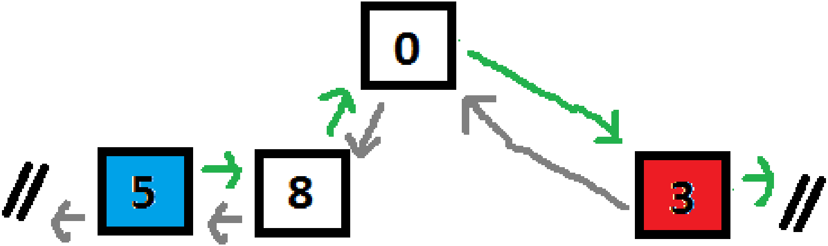
linked list: the linked list stores a pointer to the first and last nodes.

node: a node stores a data value (in this example, an **int**) and pointers to its previous and next node “neighbours”.

iterator: an iterator stores a pointer to its current node and either stores a pointer to its container or is part of its container class.

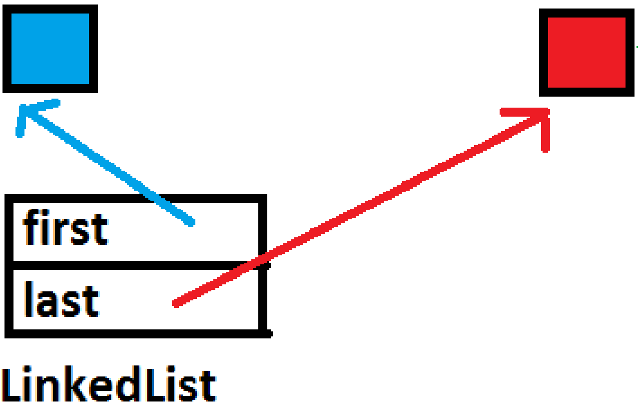
Linked Lists

From the perspective of the linked list itself, the structure is quite barren.



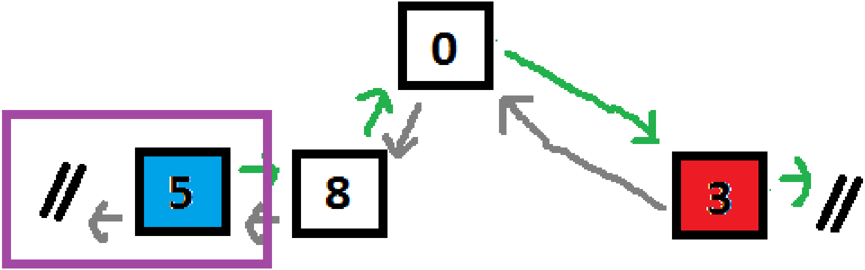
Linked Lists

From the perspective of the linked list itself, the structure is quite barren.



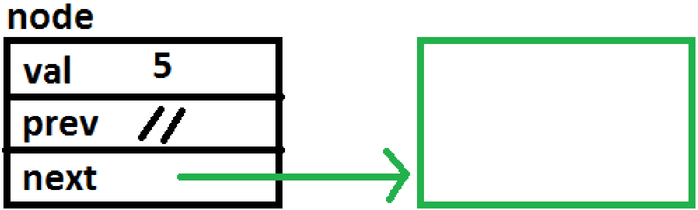
Linked Lists

A node actually stores the data for the linked list and each node must know where its previous and next nodes are, otherwise it's impossible to retrieve information from the list. Below we see three member variables, **val**, the data being stored at the node; **prev**, the previous node; and **next**, the next node.



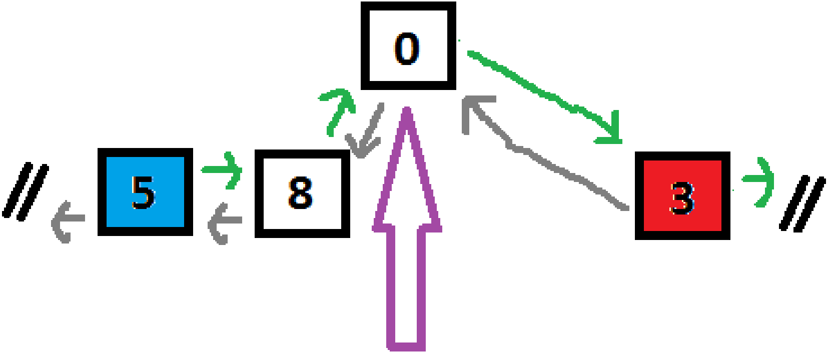
Linked Lists

A node actually stores the data for the linked list and each node must know where its previous and next nodes are, otherwise it’s impossible to retrieve information from the list. Below we see three member variables, **val**, the data being stored at the node; **prev**, the previous node; and **next**, the next node.



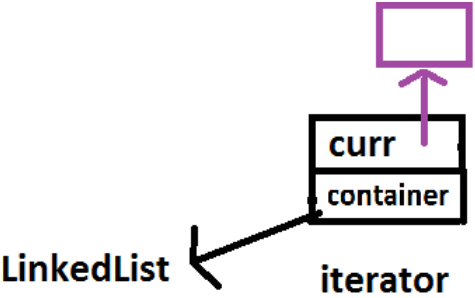
Linked Lists

An iterator is effectively a class that wraps around a node pointer. It stores a pointer **curr** to its currently managed node and a pointer to its LinkedList **container**.



Linked Lists

An iterator is effectively a class that wraps around a node pointer. It stores a pointer **curr** to its currently managed node and a pointer to its LinkedList **container**.



Linked List Interface I

The interface for the linked list is below, defined in a **intList** namespace.

```
namespace intList {
class node; // declare the node
class iterator; // declare a normal iterator
class const_iterator; // declare an iterator to work on const LinkedLists

class LinkedList{

   friend iterator; // iterators need to know first element
   friend const_iterator;

   friend void swap(LinkedList&, LinkedList&); // to swap two LinkedLists

private:
   node *first, *last; // pointers to first and last nodes
```

Linked List Interface II

```
public:
   LinkedList() noexcept; // default constructor

   LinkedList(const LinkedList&); // copy constructor

   LinkedList(LinkedList&&) noexcept; // move constructor

   ~LinkedList(); // destructor

   LinkedList& operator=(LinkedList) &; // assignment operators
```

Linked List Interface III

```
  const_iterator begin() const; // obtain const begin iterator
  const_iterator end() const; // obtain const iterator pointing past end
  iterator begin(); // obtain begin iterator
  iterator end(); // obtain iterator pointing past end

  void insert(iterator, int); // insert value before current position
  void erase(iterator); // erase value at position

  void push_back(int); // append to end
  void pop_back(); // remove from the end

  void push_front(int); // add to beginning
  void pop_front(); // remove from beginning
};
}
```

Linked List Interface

Just as a function or operator can be a **friend** of a class, so, too, can other classes. The **LinkedList** grants friendship to the **iterator** and **const_iterator** classes because the iterators may need to know where the first/last element is, depending upon its implementation.

Friendship is not automatically reciprocated: if A grants friendship to B, that does not mean B grants friendship to A!

We can add or remove elements from the back (and also the front).

Linked List Interface

The **begin** member function should return an iterator type referencing the first node of the list; the **end** member function will return an iterator referencing a null node via **nullptr** to indicate we are past the end.

Other conventions are possible for how to signify past-the-end.

Note that **begin** and **end** have been **overloaded on const**.

Linked List Interface

We intend for the **iterator** class to have the power to modify the node values upon dereferencing (by returning a reference to the node data), thus changing the list, and the **const_iterator**s to merely return by value (or reference to const) when dereferenced.

Things would otherwise be weird...

- **begin** and **end** could not be called on **const LinkedList** objects so moving through such lists would be impossible,
- or else if they were marked as accessor functions, it would be impossible to change a **LinkedList** object through the iterator type returned from **begin** and **end**.

It would be best if such weirdness did not happen...

Linked List Interface

The **std::swap** function invokes the move constructor and move assignment operator of the types being swapped if available: otherwise it copies.

We'll write our own **swap** function, not part of the **std** namespace, to swap two **LinkedList**s. We can use it in our assignment operators using the **copy-and-swap** idiom.

We'll also be giving the user the ability to swap two of our **LinkedList**s directly without an extra temporary variable.

Linked List Interface

We mark the move constructor and default constructors as **noexcept** as they cannot throw exceptions (no risk of memory exhaustion or complicated logical processes).

The copy constructor is **not marked noexcept** because it does involve memory allocation. As a result, we cannot mark the assignment operators implemented via the copy-and-swap idiom as **noexcept** because they may need to invoke the copy constructor.

If we avoided the fancy single assignment operator, we could have signatures:

```
LinkedList& operator=(const LinkedList&) &;
LinkedList& operator=(LinkedList&&) & noexcept;
```

Linked List Node Interface

```
namespace intList {
class node{

friend LinkedList; /* LinkedList needs to construct nodes and to
   know next/prev of node for insertions, deletions, etc. */

friend iterator; // iterators need to know next/prev to move, etc.
friend const_iterator;

private:

   int val; // the data
   node *prev, *next; // previous and next nodes

   node(int i); // constructor to create new node is PRIVATE
};
}
```

Linked List Node Interface

The node grants friendship to **LinkedList** because during operations such as insertion, the pointers of a node may be updated and new nodes must be created, i.e., have their constructors invoked.

The node grants friendship to **iterator** because an iterator needs to be dereferenced (and retrieve the **val** of the node) and during incrementation/decrementation, an iterator needs to know what node is next or previous.

Linked List Node Interface

The constructor sets the data variable and sets the pointers to null. It is private because no user should be constructing a **node** object themselves. It is only for the list. Likewise for the iterators.

Linked List Iterator Interface I

```
namespace intList {
class iterator{
// LinkedList may change data of iterator during operations
friend LinkedList;

public:
   iterator& operator++(); // prefix ++
   iterator operator++(int); // postfix ++

   iterator& operator--(); // prefix - -
   iterator operator--(int); // postfix - -

   int& operator*() const; // dereferencing operator (unary)

   friend bool operator==(const iterator&, const iterator&); // comparison
```

Linked List Iterator Interface II

```
private:
   node *curr; // currently managed node
   const LinkedList *container; // the iterator should not change the list

   // constructor: requires a node and list to point to
   iterator(node*, const LinkedList*);

};

// declared for fully qualified lookup: see ADL discussion
bool operator==(const iterator&, const iterator&);
// != defined by negating ==
bool operator!=(const iterator&, const iterator&);
}
```

Linked List Iterator Interface

The interface for **const_iterator** would be virtually identical, but the dereferencing operator should have signature **int operator*() const** or **const int& operator*() const** instead.

The **iterator** grants friendship to **LinkedList** because during insertions/removals, it can be relevant for the list to know where in the list an iterator is pointing.

Linked List Iterator Interface

The dereferencing function for **iterator** can be marked as **const** despite returning **int&** because the actual member variables of the **iterator** class, i.e., pointers to its node and container, are not modified even if the **int** is!

By marking the function as const, it turns the member **curr**into **node * const**, i.e., constant pointers, *not* pointers to const.

Linked List Iterator Interface

The increment and decrement operators advance the iterator forward or backwards in the list. There is a dereferencing operator so that if **itr** is an iterator referencing an element then ***itr** gives direct access to the data.

We also have a comparison operator to determine if two iterators are pointing to the same node. The **operator!=** in combination with the other functions allows us to have nice loops like

```
for(auto itr = list.begin(), end = list.end(); itr != end; ++itr) {
   std::cout << *itr << " ";
}
```

Enabling Range-Fors

For a given class **X**, there are two options we can choose between to allow traversal of our class with a **range based for loop**:

- write **begin** and **end** *member functions* that returns an iterator-type object; or if not this then
- write **begin** and **end** *free functions*, i.e., non-member functions, that accept **X**'s.

Since we have the right member functions to return iterators and our iterators have the right functionality, we can also write:

```
for (int i : list) {
   std::cout <<i <<" ";
}
```

Argument Dependent Lookup

Argument Dependent Lookup (ADL): The compiler can deduce the namespace of a free function by its arguments.

Strictly speaking we declared
bool intList::operator==(const intList::iterator&,
const intList::iterator&)
and one might imagine that, in code, we would need to write:

```
intList::operator==(it1, it2); // this instead of:

// it1 == it2
```

however, for **it1** and **it2** of type **intList::iterator**, because the arguments are part of the **intList** namespace, the proper operator can be invoked with just

```
it1 == it2
```

Argument Dependent Lookup

As another example:

```
// no using namespace std has been written!

std::vector<int> v {1,2,3};
auto it = std::begin(v); // okay, obviously!
auto it2 = begin(v); // okay, surprisingly!

std::cout << std::endl; // okay, obviously
endl(std::cout); // okay, surprisingly!

std::cout << endl; // ERROR: endl needs scope
```

Argument Dependent Lookup

Because **v** is of type **std::vector<int>**, belonging to the **std** namespace, when **begin(v)** is called, a **begin** function can be looked up within the **std** namespace.

Because **std::endl** is a function, even without its fully qualified **std::** name, it can be inferred because **std::cout** is in the **std** namespace and there is an **std::endl** function accepting **std::ostream&**.

In the case that fails, the identity of **endl** cannot be deduced as it isn't called as a function.

Argument Dependent Lookup

If a friend declaration in a non-local class first declares a class, function, class template or function template the friend is a member of the innermost enclosing namespace. The friend declaration does not by itself make the name visible to unqualified lookup or qualified lookup. [Note: The name of the friend will be visible in its namespace if a matching declaration is provided at namespace scope (either before or after the class definition granting friendship). —end note] If a friend function or function template is called, its name may be found by the name lookup that considers functions from namespaces and classes associated with the types of the function arguments.

Argument Dependent Lookup

Remark: as a consequence, **in addition to declaring a friend function within the class, it should also be declared without the class, but within the scope of the namespace** if we want to allow explicit name lookup with **::** and lookup via ADL.

For example, we should also declare:

```
void swap(LinkedList&, LinkedList&);
```

inside the namespace of **intList.** Then someone can also invoke the function via **intList::swap(L1,L2);**, etc.

Linked List Implementations

For the implementations, we suppose they are given in a **.cpp** file and writing the definitions within the **namespace intList { ... }**.

Constructors:

```
// empty list with null first and last
LinkedList::LinkedList() noexcept : first(nullptr), last(nullptr) {}

// node stores value but points to null on either side
node::node(int _val) : val(_val), prev(nullptr), next(nullptr) {}

// iterator references node n for list ell
iterator::iterator(node* n, const LinkedList* ell) : curr(n), container(ell) {}
```

Linked List Implementations

The swap:

```
// swap directly swaps the pointers
void swap(LinkedList& one, LinkedList& another) {
   std::swap(one.first, another.first);
   std::swap(one.last, another.last);
}
```

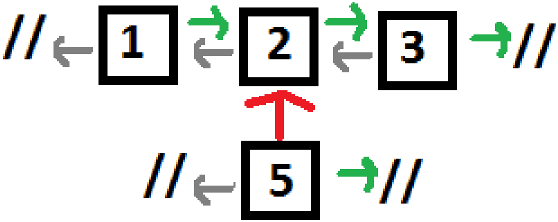
Since **swap** is a friend, it can access/modify the contents of **LinkedList** classses.

Linked List Implementations: Insertions/Removals

Having already seen the natural growth of the list, we look at insertion and removal processes for a list.

Note the process may depend upon the location, e.g., removing at the middle vs end, etc.

Want to insert new node with 5 before node with 2.

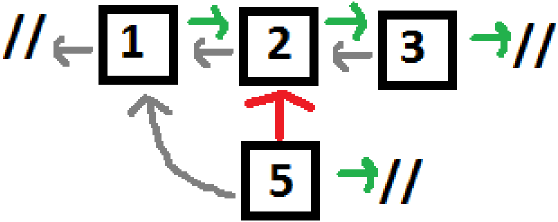


Linked List Implementations: Insertions/Removals

Having already seen the natural growth of the list, we look at insertion and removal processes for a list.

Note the process may depend upon the location, e.g., removing at the middle vs end, etc.

Set new node's **prev** pointer.

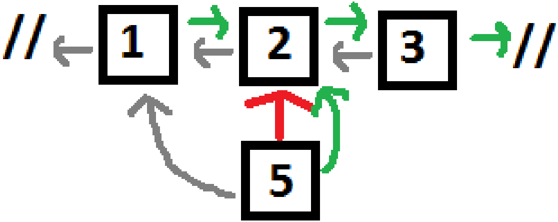


Linked List Implementations: Insertions/Removals

Having already seen the natural growth of the list, we look at insertion and removal processes for a list.

Note the process may depend upon the location, e.g., removing at the middle vs end, etc.

Set new node's **next** pointer.

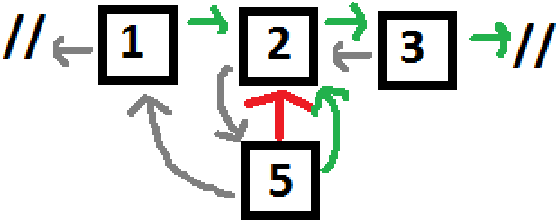


Linked List Implementations: Insertions/Removals

Having already seen the natural growth of the list, we look at insertion and removal processes for a list.

Note the process may depend upon the location, e.g., removing at the middle vs end, etc.

Set the new node's next node's **prev** pointer to point to new node.

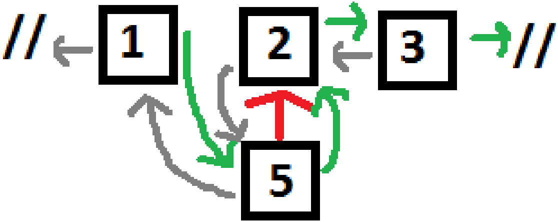


Linked List Implementations: Insertions/Removals

Having already seen the natural growth of the list, we look at insertion and removal processes for a list.

Note the process may depend upon the location, e.g., removing at the middle vs end, etc.

Set the new node's previous node's **next** pointer to point to new node.

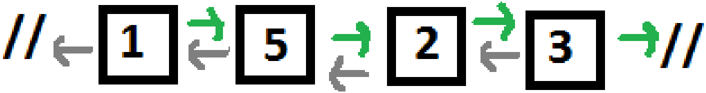


Linked List Implementations: Insertions/Removals

Having already seen the natural growth of the list, we look at insertion and removal processes for a list.

Note the process may depend upon the location, e.g., removing at the middle vs end, etc.

The process is complete.

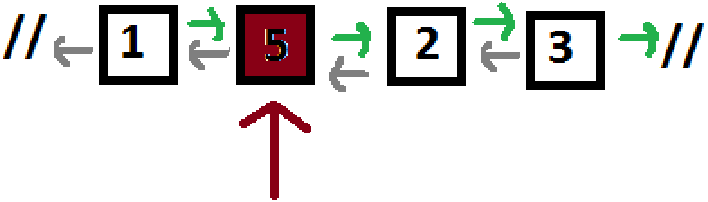


Linked List Implementations: Insertions/Removals

Having already seen the natural growth of the list, we look at insertion and removal processes for a list.

Note the process may depend upon the location, e.g., removing at the middle vs end, etc.

Want to remove a node.

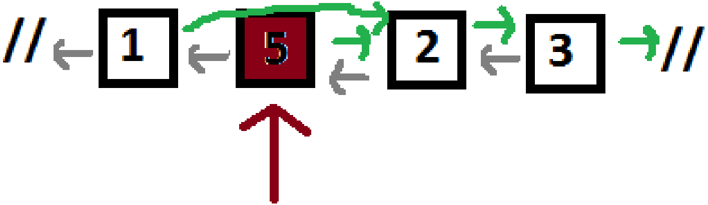


Linked List Implementations: Insertions/Removals

Having already seen the natural growth of the list, we look at insertion and removal processes for a list.

Note the process may depend upon the location, e.g., removing at the middle vs end, etc.

Set its previous node to point to its **next** node.

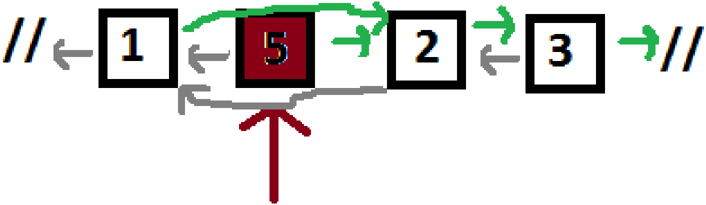


Linked List Implementations: Insertions/Removals

Having already seen the natural growth of the list, we look at insertion and removal processes for a list.

Note the process may depend upon the location, e.g., removing at the middle vs end, etc.

Set its next node to point to its **prev** node.

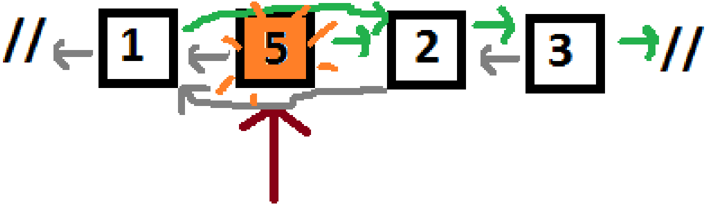


Linked List Implementations: Insertions/Removals

Having already seen the natural growth of the list, we look at insertion and removal processes for a list.

Note the process may depend upon the location, e.g., removing at the middle vs end, etc.

Destroy the node.

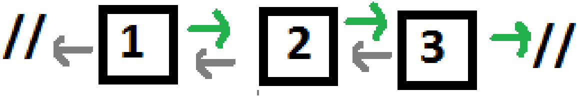


Linked List Implementations: Insertions/Removals

Having already seen the natural growth of the list, we look at insertion and removal processes for a list.

Note the process may depend upon the location, e.g., removing at the middle vs end, etc.

The process is complete.

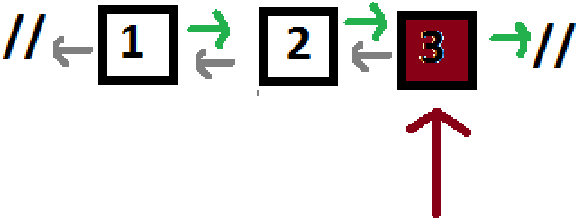


Linked List Implementations: Insertions/Removals

Having already seen the natural growth of the list, we look at insertion and removal processes for a list.

Note the process may depend upon the location, e.g., removing at the middle vs end, etc.

Want to remove node at end.

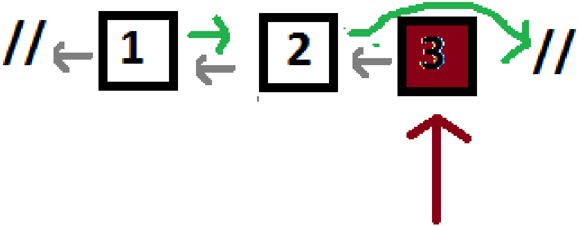


Linked List Implementations: Insertions/Removals

Having already seen the natural growth of the list, we look at insertion and removal processes for a list.

Note the process may depend upon the location, e.g., removing at the middle vs end, etc.

Set its previous node's **next** to point to **null**.

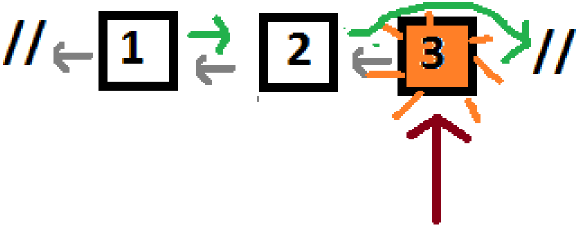


Linked List Implementations: Insertions/Removals

Having already seen the natural growth of the list, we look at insertion and removal processes for a list.

Note the process may depend upon the location, e.g., removing at the middle vs end, etc.

Destroy the node.

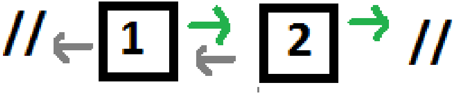


Linked List Implementations: Insertions/Removals

Having already seen the natural growth of the list, we look at insertion and removal processes for a list.

Note the process may depend upon the location, e.g., removing at the middle vs end, etc.

The process is complete.



Linked List Implementations: Insertions/Removals

```
void LinkedList::push_back(int val){

    node *n = new node(val); // create a new node

    if (last) { // if last node not null
        last->next = n; // make last point to n
        n->prev = last; // make n have last as its previous
        last = n; // update the last position
    }
    else { // in this case the last node is null so list is empty
        first = last = n; // both equal to n
    }
}
```

Linked List Implementations: Insertions/Removals

```
void LinkedList::pop_back(){

   if (!last) { // list is empty if last null
      throw std::logic_error("pop back on empty list");
   }
   else if (first == last){ // if just one element
      delete first; // free heap memory
      first = last = nullptr; // make both null because empty again
   }
   else { // many elements
      node *newlast = last->prev; // store the new last node
      newlast->next = nullptr; // set its next pointer to null
      delete last; // free the heap memory
      last = newlast; // update the last value
   }
}
```

Linked List Implementations: Insertions/Removals I

```
void LinkedList::insert(iterator it, int val) {
    if (it.curr == nullptr) { // then inserting before past-the-end
        push_back(val);
    }

    else if (it.curr == first) { // then at beginning
        push_front(val);
    }
```

Linked List Implementations: Insertions/Removals II

```
  else{ // then in middle

     node *n = new node(val); // create new node
     n->next = it.curr; // set n's next
     n->prev = it.curr->prev; // set n's previous

     it.curr->prev->next = n; // make current previous' next node into n
     it.curr->prev = n; // make current previous node into n

  }
}
```

Linked List Implementations: Insertions/Removals

```
void LinkedList::erase(iterator it) {
   if (it.curr == first) { // removing first
      pop_front(); // remove first
   }
   else if (it.curr == last) { // removing last
      pop_back(); // remove last
   }
   else { // somewhere in the middle
      it.curr->prev->next = it.curr->next; // reroute next of previous
      it.curr->next->prev = it.curr->prev; // reroute previous of next
      delete it.curr; // free the heap memory of item being removed
   }
}
```

Linked List Implementations: Copy and Move Constructors

```
// copy constructor: copy elements over one by one
LinkedList::LinkedList(const LinkedList& rhs) : first(nullptr), last(nullptr) {
   for (int i : rhs) { // take each value from rhs
      push_back(i);
   }
}

/* move constructor: take pointers to first and last then set rhs pointers to
   null so it is in valid destructible state */
LinkedList::LinkedList(LinkedList&& rhs) noexcept : LinkedList() { //
default
   swap(*this, rhs); // use member swap
}
```

Linked List Implementations: Destructor

```
LinkedList::~LinkedList() {
    node *n = first; // start at first node
    while (n != nullptr) { // while not past the end denoted by nullptr
        node *temp = n->next; // temporarily store the next position
        delete n; // delete the node on heap
        n = temp; // move n right
    }
}
```

Linked List Iterator Implementations: Prefix Decrement

```
iterator& iterator::operator--(){
   if(curr == container->first) { // cannot go before first
      throw std::logic_error("Invalid address");
   }
   else if(curr == nullptr) { // just past the end, go to last
      curr = container->last; // now make iterator refer to last element
   }
   else { // in the middle somewhere
      curr = curr->prev; // reference previous node
   }
   return *this;
}

iterator iterator::operator--(int) {
   iterator copy(*this);
   --(*this); // or this->operator--();
   return copy;
}
```

Linked List Iterator Implementations: Dereferencing and Comparison Operators

```
// dereferencing operator for iterator
int& iterator::operator*() const {
   return curr->val; // return reference to the int stored
}

// dereferencing operator for const_iterator
int const_iterator::operator*() const {
   return curr->val; // return copy of the int stored
}

// comparison for iterator
bool operator==(const iterator& left, const iterator& right) {
   return ( (left.curr == right.curr) && (left.container == right.container) );
}
```

Costing for Linked Lists

One of the advantages of linked a list over a data structure such as a vector is the $O(1)$ insertion/deletion time (one just needs to move around some pointers), whereas with a vector that cost is in general $O(n)$ (some or all elements may need to be shifted).

In reality, though, this is seldom an advantage because looking up an item in a linked list is $O(n)$: one can start at the beginning or the end and move through sequentially.

Linked lists *do not support random access*: being able to access an item at position **lst.begin() + 7** directly, for example. We have to iterate forwards from the first element. This is because the data are not contiguous.

Nested Classes

Before writing other container classes, we will consider **nested classes**.

Just as member variables are stored within a class a class can have member functions, a class can have “member classes”.

These are useful in hiding implementation details from users. For example, with the **LinkedList** class, a user of the class should never need to define or create a **node** object and it would make sense to hide that from users by securing the **node** class as a private component of the **LinkedList**.

Nested Classes

Nested classes also allow for a protection of the namespace one is working in; and the additional scoping involved with nesting allows for two classes to be tied together very closely. Think of **std::vector<int>::iterator**: a vector of **int** iterator only makes sense as a helper class for a **std::vector<int>**, etc.

Implementing nested classes is very similar to normal classes except that one needs to specify the scope of the surrounding class when defining/implementing inner classes outside of the outer class.

Nested Classes

When nesting classes, *all classes have a separate, independent existence of each other.* An inner class could exist on its own, without the existence of an outer class, etc.

The *outer class automatically grants friendship to the inner class.* Therefore an inner class can access the private member variables of a variable of its outer class type.

Nested Classes

```
class Out1 {
private:
   int secret = 42;
public:
   struct In1 { // define Out1::In1
      void add(Out1& out) const { ++out.secret; }
   };
};
```

Above **Out1** is defined, as is **Out1::In1.** Both classes can exist independently but **In1** is a friend to **Out1**:

```
Out1 x; // create Out1
Out1::In1 y; // create Out1::In1 class - scope Out1:: required

y.add(x); // the private secret of x is now 43
```

Nested Classes

```
class Out2 {
private:
   class In2; // declare Out2::In2
public:
   In2 get() const; // declare accessor
};

class Out2::In2 { // define Out2::In2
public:
   void hi() const { std::cout << "hi!"; }
};

Out2::In2 Out2::get() const { return In2(); } // define accessor
```

Nested Classes

In2 is **private** within **Out2** so this doesn't work:

```
Out2::In2 x; // ERROR: In2 is private - cannot reference that name!
```

but we can obtain an **In2** with **auto**:

```
auto x = Out2().get(); // returns an Out2::In2 object captured by auto
x.hi();
```

Nested Classes

Declarations and definitions: we can only define **Out2::get** after **Out2::In2** because **Out2::get** invokes the default constructor of **Out2::In2** and the compiler doesn't know that exists until after the **Out2::In2** class has been defined!

Scope: note that we can simply **return In2();** without scoping it as **return Out2::In2();** because we are providing that implementation within the **Out2** scope (it is a member function of **Out2**!).

Binary Search Trees

A binary search tree allows for data to be stored in sorted order. Like a linked list, the data are stored in nodes and are not stored contiguously. Unlike a linked list, there are precise rules for how data are inserted into the tree based on their “size”.

A binary search tree stores a node as its root. Each node in the tree can have two child nodes that it points to: a left node (a node storing a smaller value) and a right node (a node with a larger value). Some implementations also allow for nodes to store a pointer to their parent node.

Binary Search Trees

We consider designing a **Tree** class that acts as a binary search tree storing **std::string** objects. We will include nested **node** and **iterator** classes.

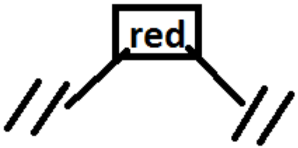
We will define all of this within the a namespace called **stringBinTree**.

Binary Search Tree Visual

We consider adding the strings (in order): “red”, “orange”, “yellow”, “green”, “blue”, “indigo”, and “violet”.

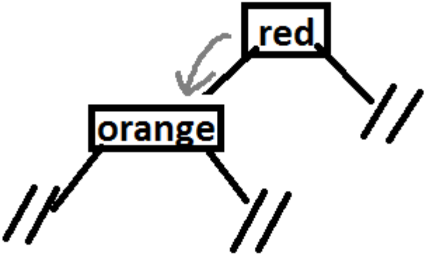
Binary Search Tree Visual

We consider adding the strings (in order): “red”, “orange”, “yellow”, “green”, “blue”, “indigo”, and “violet”.



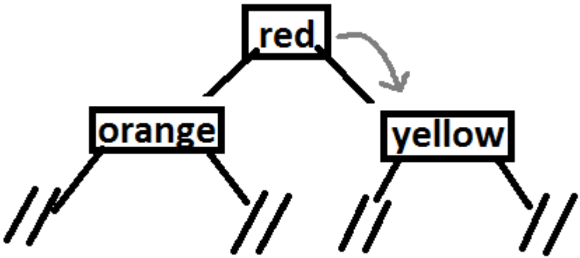
Binary Search Tree Visual

We consider adding the strings (in order): “red”, “orange”, “yellow”, “green”, “blue”, “indigo”, and “violet”.



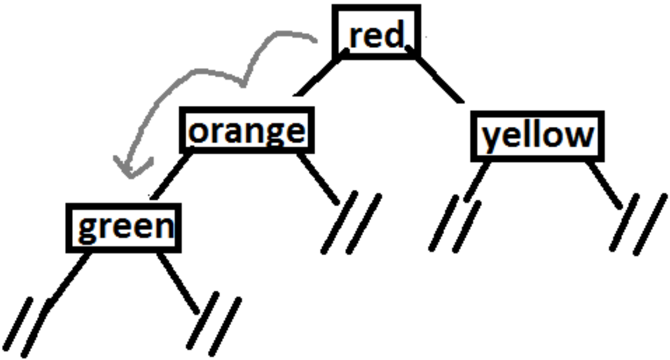
Binary Search Tree Visual

We consider adding the strings (in order): “red”, “orange”, “yellow”, “green”, “blue”, “indigo”, and “violet”.



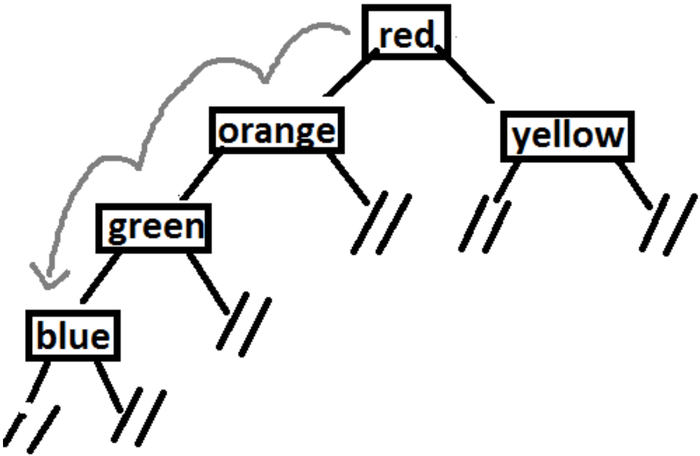
Binary Search Tree Visual

We consider adding the strings (in order): “red”, “orange”, “yellow”, “green”, “blue”, “indigo”, and “violet”.



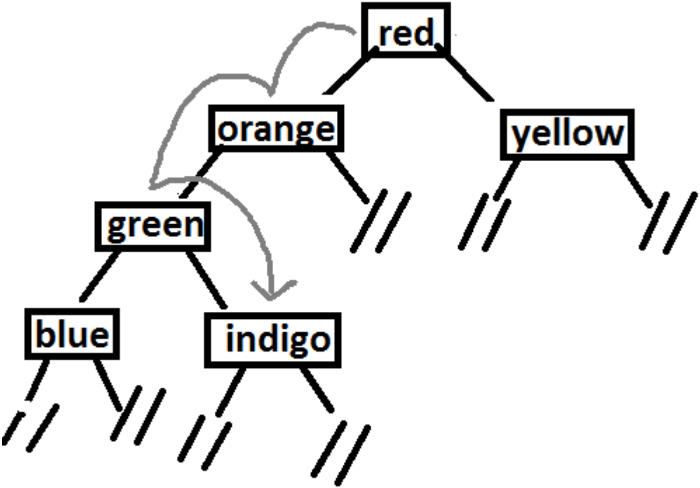
Binary Search Tree Visual

We consider adding the strings (in order): “red”, “orange”, “yellow”, “green”, “blue”, “indigo”, and “violet”.



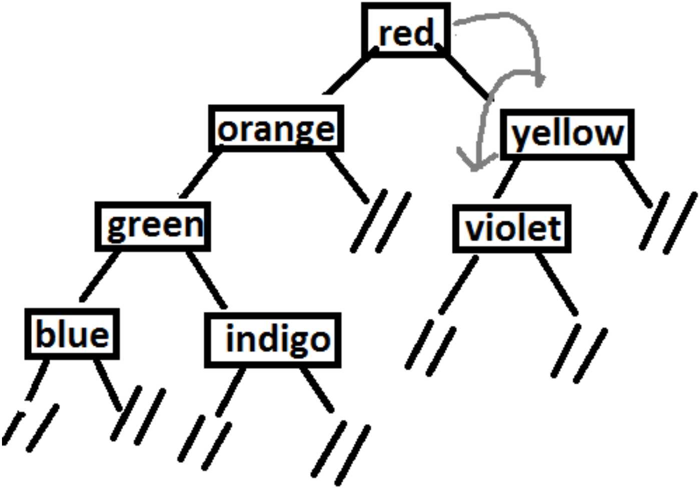
Binary Search Tree Visual

We consider adding the strings (in order): “red”, “orange”, “yellow”, “green”, “blue”, “indigo”, and “violet”.



Binary Search Tree Visual

We consider adding the strings (in order): “red”, “orange”, “yellow”, “green”, “blue”, “indigo”, and “violet”.



Binary Search Tree Implementation

Each time a new item is added to the tree, its value is compared against the root to determine if it should branch left (if <) or right (if >).

The comparison process is repeated with the next nodal value until it takes a branch with a null pointer in which case the new node is placed in lieu of the null pointer.

This is a recursive process!

Binary Search Tree Interface I

```
namespace stringBinTree {
   class Tree {

   private:
      class node; // nested node class

      node *root; // the root of the tree
      void deleteTree(node*); // to recursively delete the tree
      void traverseInsert(node*); // to help with copying
```

Binary Search Tree Interface II

```
public:
    class const_iterator; // nested iterator class

    Tree() noexcept; // default constructor for empty tree
    ~Tree(); // destructor
    Tree(const Tree&); // copy constructor
    Tree(Tree&&) noexcept; // move constructor
    Tree& operator=(Tree) &; // assignment operators

    bool find(const std::string&) const; // check if contains a string

    friend void swap(Tree&, Tree&); // swap two Trees
```

Binary Search Tree Interface III

```
    const_iterator begin() const; // iterator to begin position
    const_iterator end() const; // iterator to past-the-end position

    void insert(std::string); // to add a value to the tree
    void erase(const_iterator); // to erase a value from the tree
};

void swap(Tree&,Tree&); // declare at namespace level, too
bool operator==(const Tree::const_iterator&,
    const Tree::const_iterator&);
bool operator!=(const Tree::const_iterator&,
    const Tree::const_iterator&);
```

Binary Search Tree Node Interface

```
class Tree::node {

friend Tree; // tree member functions may search through nodes
friend const_iterator; // to be able to advance by checking node values

private:
   node(std::string); // constructor: take by value and move it

   node *left, *right; // children
   std::string val; // data value stored
   void insertNode(node*); // member function for inserting node
};
```

Binary Search Tree Node Interface

Note that within **Tree::node**, we can refer to **Tree::const_iterator** as just **const_iterator** because by working within **Tree::node**, we are within the **Tree** class scope.

Binary Search Tree Iterator Interface I

```
class Tree::const_iterator {

friend Tree; // to allow iterator modifications by Tree operations

private:
   const_iterator(node*, const Tree*); // constructor

   node *curr; // current position
   const Tree *container; // holding container

public:
   const_iterator &operator++(); // prefix ++
   const_iterator operator++(int); // postfix ++
   const_iterator &operator--(); // prefix - -
   const_iterator operator--(int); // postfix - -
```

Binary Search Tree Iterator Interface II

```
    const std::string& operator*() const; // dereference operator

    const std::string* operator->() const { // arrow operator
      return & (curr->val);
    }

    friend bool operator==(const const_iterator& left,
      const const_iterator& right); // comparisons
  };

  bool operator!=(const const_iterator& left,
    const const_iterator& right);

} // ending namespace brace
```

Binary Search Tree Iterator Interface

The **operator->** is called **operator arrow** and it must always be a member function. We use this operator along with the dereferencing operator to give a class an iterator/pointer-like behaviour.

Note that they are implemented as accessor member functions instead of mutator member functions. If they allowed modifications, this could destroy the sorted structure of the tree through a command such as:

```
/* BAD: what if it pointed to "cat", with its proper place in the tree
   but we did */
it->clear(); // makes the string empty: sorting now off
```

Binary Search Tree Implementations

We now assume we define the functions in a **.cpp** file inside of the **namespace stringBinTree { ... }**.

```
void Tree::insert(std::string val) {
   if (!root) { // if the root is empty
      // make a new node and set this to be root
      root = new node( std::move(val) );
   }
   else{ // otherwise
      node *n = new node( std::move(val) ); // create a new node

      // and recursively pass it node to node until in place
      root->insertNode(n);
   }
}
```

Binary Search Tree Implementations

```
Tree::Tree(const Tree& rhs) : root(nullptr) {
   traverseInsert(rhs.root); // calls a recursive function on nodes to copy
}

Tree::Tree(Tree&& that) noexcept : Tree() {
   swap(*this, that);
}

Tree& Tree::operator=(Tree that) & {
   swap(*this, that);
   return *this;
}
```

Binary Search Tree Implementations I

```
void Tree::node::insertNode(node* n){

   // if this value is less than new node value, new node should go right
   if (val < n->val) {
      if (right) { // if this node has a right child
         right->insertNode(n); // recurse on the right child
      }
      else { // if the right child is null
         right = n; // make this the right child
      }
   }
```

Binary Search Tree Implementations II

```
    // if this value is larger than new node value, new node should go left
    else if (val > n->val) {
        if (left) { // if this node has a left child
            left->insertNode(n); // recurse on the left child
        }
        else { // if the left child is null
            left = n; // make this the left child
        }
    }
    else {
    // nothing to add if new node value neither < nor > than current value
    delete n; // but we should free the allocated node memory
    }
}
```

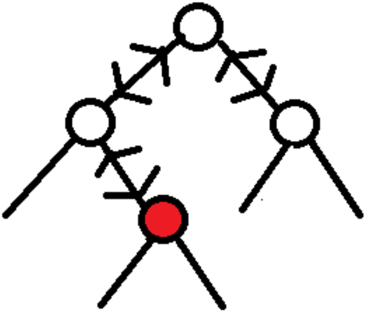
Binary Search Tree Implementations

```
Tree::const_iterator Tree::begin() const { // return type requires scope
    if (root == nullptr) { // if root is null then tree empty
        return const_iterator(nullptr, this); // return iterator that is null
    }

    node *n = root; // start at the root

    while (n->left != nullptr) { // while we can still go left (to lower value)
        n = n->left; // go left
    }

    return const_iterator(n, this); // return iterator for node of smallest value
}

// end iterator means “past the end” and should store null
Tree::const_iterator Tree::end() const{
    return const_iterator(nullptr, this);
}
```

Binary Search Tree Implementations

We consider pictorial representations for the node removal processes. In this case, it is useful for each node to also track its parent node (another member variable).

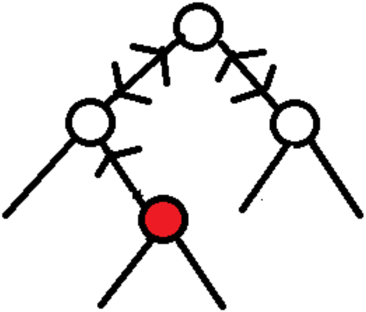
To remove a node with no children and a non-null parent...



Binary Search Tree Implementations

We consider pictorial representations for the node removal processes. In this case, it is useful for each node to also track its parent node (another member variable).

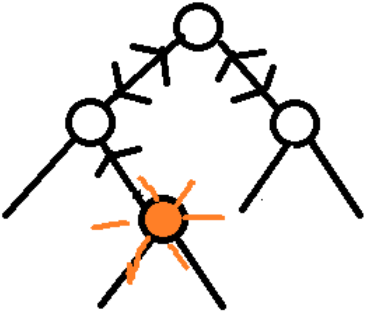
Set its parent's pointer that pointed to the node to null



Binary Search Tree Implementations

We consider pictorial representations for the node removal processes. In this case, it is useful for each node to also track its parent node (another member variable).

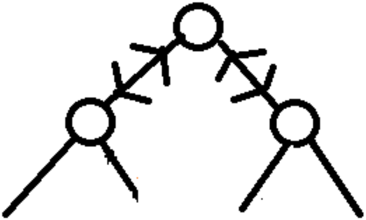
Delete the node



Binary Search Tree Implementations

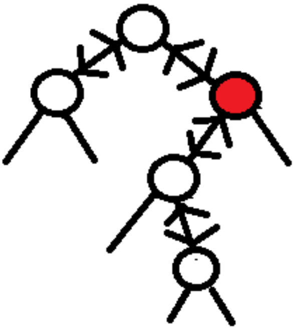
We consider pictorial representations for the node removal processes. In this case, it is useful for each node to also track its parent node (another member variable).

Process done



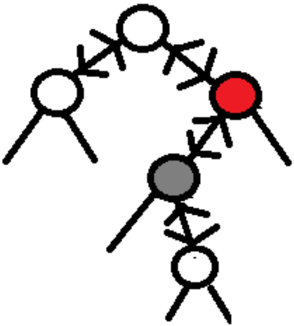
Binary Search Tree Implementations

To remove a given node with one child and the given node has a parent ...



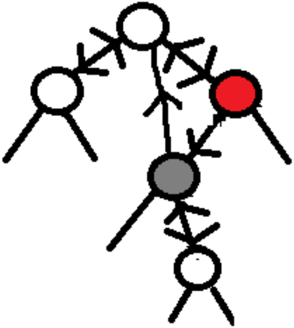
Binary Search Tree Implementations

Go to the child of the given node



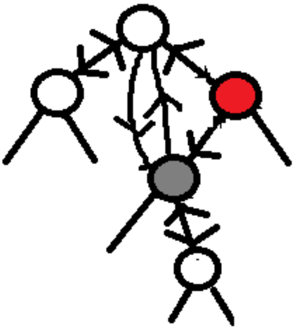
Binary Search Tree Implementations

Set given node's child's parent to given node's parent



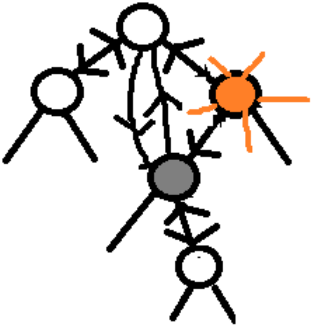
Binary Search Tree Implementations

Set given node's parent's corresponding child to given node's child.



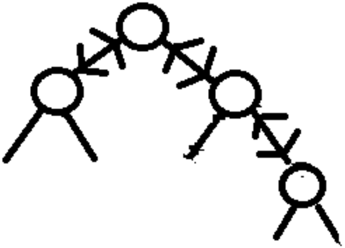
Binary Search Tree Implementations

Delete the node



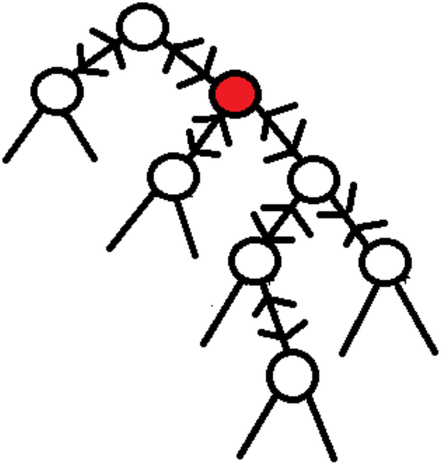
Binary Search Tree Implementations

Process done



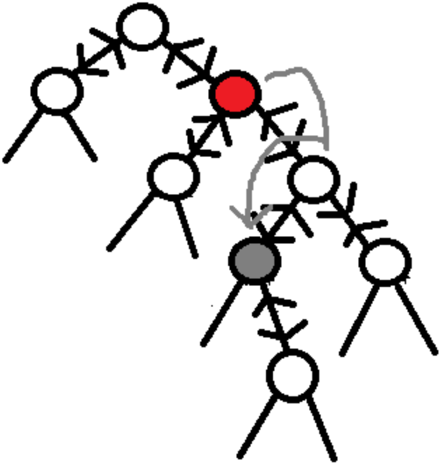
Binary Search Tree Implementations

To remove a given node with two children...



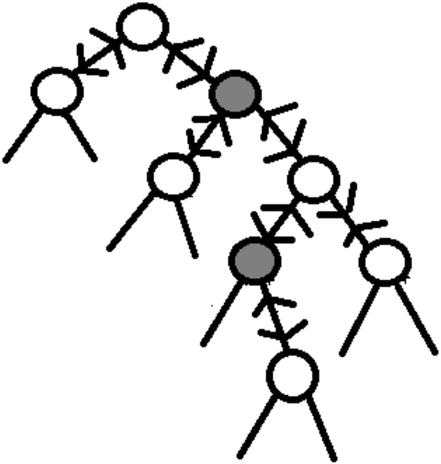
Binary Search Tree Implementations

Move right from it and as far left as possible



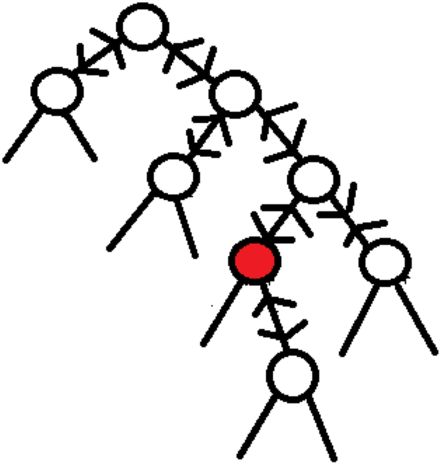
Binary Search Tree Implementations

Overwrite the given node's value with the right-far-left value



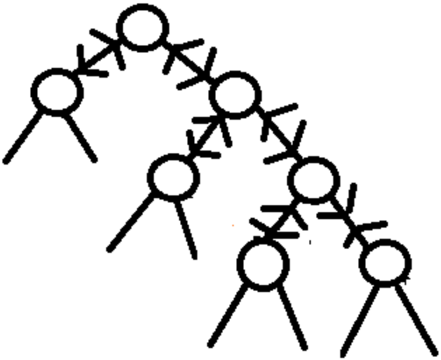
Binary Search Tree Implementations

Remove the right-far-left node with zero or one children



Binary Search Tree Implementations

Process done



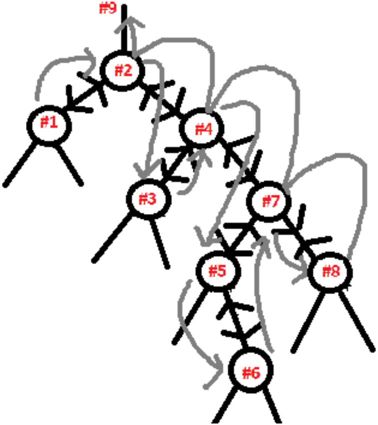
Binary Search Tree Implementations

Remark: removal is recursive but a very simple recursion. We either start in the base case with a node having zero or one children, or if the node has two children, we immediately reduce to that base case.

Binary Search Tree Implementations

To move forward from one node to the next, we look for a right child and if there is one, then move to it and go as far left as possible.

If there is no right child, we move up through the parent nodes until one of those upward moves takes us farther right.



Binary Search Tree Implementation Remarks

The **traverseInsert** function to assist with copying could follow a recursive pattern: given a node **n** (initially the copy-from tree's **root**), if it is not null:

- **insert(n->val)**;
- if **n->left** is not null, **traverseInsert(n->left)** or else do nothing; then,
- if **n->right** is not null, **traverseInsert(n->right)** or else do nothing.

A similar idea holds with the destructor.

Being able to traverse a tree with an **iterator** can be made a lot easier if each node stores a pointer to its **parent** node.

Being able to track the number of elements can be done very easily by having **insert** return a **bool**, depending on whether an insertion happens and incrementing the size if **true**.

Binary Search Tree Costing

A binary search tree can be a highly efficient data structure.

At worst, a binary search tree could behave as a linked list: if the data are added in sorted order: it would form a long chain-like structure.

It is not recommended to use a binary search tree on data that is already sorted. On the other hand, most of the time the data come in randomly enough that the tree is **balanced** and many nodes do have two children, etc.

Binary Search Tree Costing

Insertion for a balanced binary search tree is $O(\log n)$: $O(\log n) + O(1)$ where the first term comes from finding the position through “divide and conquer” and the second term is the process of appending a node.

Deletion is $O(\log n)$ for our erase process.

As with linked lists, *binary search trees do not support random access*.

Binary Search Tree Costing

An **std::set** is more efficient in its cost as it is guaranteed to be $O(\log n)$ even if the data come in sorted.

Typically an **std::set** is built upon a binary search tree with additional rules imposed to ensure the tree stays balanced.

Vector

Recall that a vector stores data contiguously on the heap although the C++ Standard formally requires that the vector have iterators obeying:

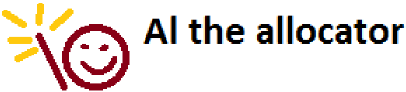
*"... for integral values n and dereferenceable iterator values a and (a + n), *(a + n) is equivalent to *(addressof(*a) + n)"*

In a sense this is more general than contiguous, although contiguous memory blocks also fit this requirement.

Vector

As more space is needed, elements are moved to the new location for further additions to the vector.

Internally, a vector manages its space by means of an **allocator**. The **std::allocator** is a templated class that can create blocks of unconstructed memory for a given type, place/construct objects, destroy objects, and free segments of memory.

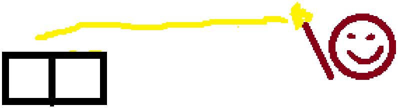


Vector

As more space is needed, elements are moved to the new location for further additions to the vector.

Internally, a vector manages its space by means of an **allocator**. The **std::allocator** is a templated class that can create blocks of unconstructed memory for a given type, place/construct objects, destroy objects, and free segments of memory.

Memory allocation



size = 0   capacity = 2

Vector

As more space is needed, elements are moved to the new location for further additions to the vector.

Internally, a vector manages its space by means of an **allocator**. The **std::allocator** is a templated class that can create blocks of unconstructed memory for a given type, place/construct objects, destroy objects, and free segments of memory.

Construction

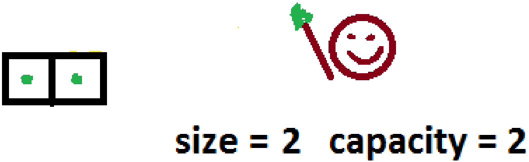
size = 1 capacity = 2

Vector

As more space is needed, elements are moved to the new location for further additions to the vector.

Internally, a vector manages its space by means of an **allocator**. The **std::allocator** is a templated class that can create blocks of unconstructed memory for a given type, place/construct objects, destroy objects, and free segments of memory.

Construction

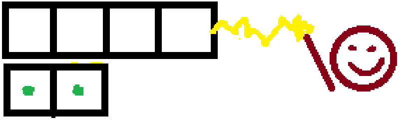


Vector

As more space is needed, elements are moved to the new location for further additions to the vector.

Internally, a vector manages its space by means of an **allocator**. The **std::allocator** is a templated class that can create blocks of unconstructed memory for a given type, place/construct objects, destroy objects, and free segments of memory.

Memory allocation



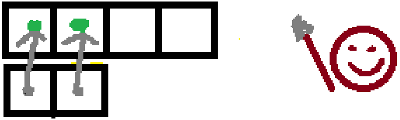
size = 2 capacity = 2

Vector

As more space is needed, elements are moved to the new location for further additions to the vector.

Internally, a vector manages its space by means of an **allocator**. The **std::allocator** is a templated class that can create blocks of unconstructed memory for a given type, place/construct objects, destroy objects, and free segments of memory.

Moving elements over



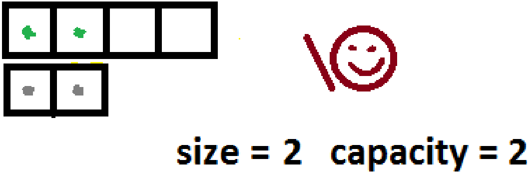
size = 2 capacity = 2

Vector

As more space is needed, elements are moved to the new location for further additions to the vector.

Internally, a vector manages its space by means of an **allocator**. The **std::allocator** is a templated class that can create blocks of unconstructed memory for a given type, place/construct objects, destroy objects, and free segments of memory.

Memory is leftover

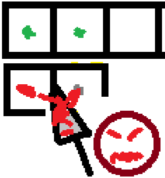


Vector

As more space is needed, elements are moved to the new location for further additions to the vector.

Internally, a vector manages its space by means of an **allocator**. The **std::allocator** is a templated class that can create blocks of unconstructed memory for a given type, place/construct objects, destroy objects, and free segments of memory.

Moved from elements destroyed



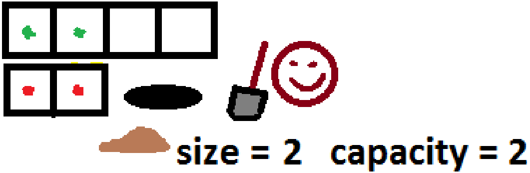
size = 2   capacity = 2

Vector

As more space is needed, elements are moved to the new location for further additions to the vector.

Internally, a vector manages its space by means of an **allocator**. The **std::allocator** is a templated class that can create blocks of unconstructed memory for a given type, place/construct objects, destroy objects, and free segments of memory.

Moved from elements destroyed

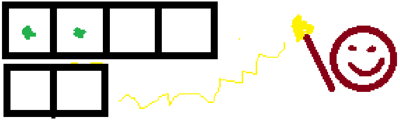


Vector

As more space is needed, elements are moved to the new location for further additions to the vector.

Internally, a vector manages its space by means of an **allocator**. The **std::allocator** is a templated class that can create blocks of unconstructed memory for a given type, place/construct objects, destroy objects, and free segments of memory.

Memory deallocation



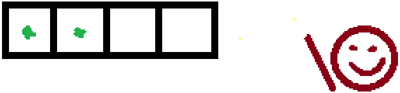
size = 2   capacity = 2

Vector

As more space is needed, elements are moved to the new location for further additions to the vector.

Internally, a vector manages its space by means of an **allocator**. The **std::allocator** is a templated class that can create blocks of unconstructed memory for a given type, place/construct objects, destroy objects, and free segments of memory.

Vector ready to grow more



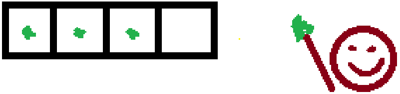
size = 2   capacity = 4

Vector

As more space is needed, elements are moved to the new location for further additions to the vector.

Internally, a vector manages its space by means of an **allocator**. The **std::allocator** is a templated class that can create blocks of unconstructed memory for a given type, place/construct objects, destroy objects, and free segments of memory.

Construction



size = 3 capacity = 4

Why Allocators?

Recall that we can use **new** to create a dynamic array. However, class objects must be default initialized.

```
std::string *sptr = new std::string[1000]; // 1000 strings of ""
```

Why Allocators?

If a class does not have a default constructor, it cannot be dynamically allocated on the heap as an array.

Also, even if the class does have a default constructor, it is inefficient to go through the process of initializing many instances of the class that will almost surely be overwritten later on.

Allocators reserve the memory for the objects/types they store, but do no initialization. They are defined with the **<memory>** header.

Using Allocators I

We consider an archetypal class **Foo:**

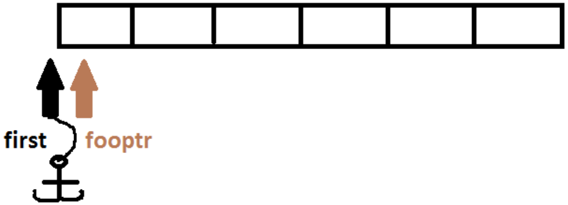
```
struct Foo {
   Foo(); // has default
   Foo(int); // can accept an int
   Foo(double,double,double); // can accept 3 doubles
   Foo(const Foo&); // copy constructor
   // ... other members and stuff...
};
```

Some of the key functionalities are illustrated below for managing a block of memory for **Foo** objects. Think "array of Foos."

Using Allocators II

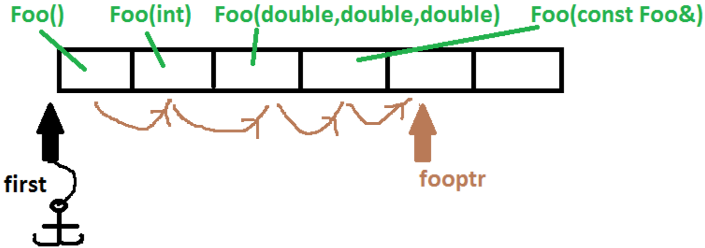
```
std::allocator< Foo > Al; // default allocator constructor

const size_t num = 6;

// allocate memory segment for 6 Foos
// get and store first position
// fptr points to first address of segment
Foo * fooptr = Al.allocate(num);
Foo * const first = fooptr; // first position cannot be modified
```

Using Allocators III



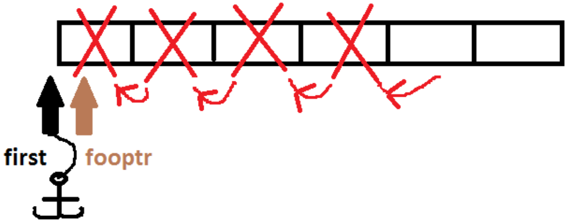
Using Allocators IV

```
// specify position and construction parameter(s), advance vptr
Al.construct(fooptr++); // default constructed Foo
Al.construct(fooptr++, 8); // Foo(int) called
Al.construct(fooptr++, 1.1, 2.2, 3.3); // Foo(double,double,double) called
Al.construct(fooptr++, *first); // Foo(const Foo&) called
```



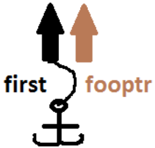
Using Allocators V

```
// give location of constructed element and destroy it
while( fooptr != first ) { // stop when point to first
    Al.destroy( -- fooptr ); // destroy elements that were created
}
```



Using Allocators VI

// free up the memory - give starting address and space used
Al.deallocate(first, num);



Using Allocators

The **construct** member function is **variadic** in that it can take variable sizes of arguments.

Beyond the first required argument of a position to construct an object, it can take zero or more input arguments that would be used to construct an object.

With zero extra arguments, a default **Foo** was constructed.
With just **8** given, this called **Foo(int)**
With **1.1,2.2,3.3** given, this called **Foo(double,double,double)**
With the l-value ***first** given, this copy-construted to form a new **Foo**.

Using Allocators

Memory cannot be dereferenced or subscripted, etc., *until it has been first constructed*. It wouldn't make sense to access an object that doesn't exist yet!

Destruction must happen prior to deallocation. Both must happen to properly free the memory.

Using an **std::allocator** is working more closely with memory than the **new** and **delete** expressions.

Vector Interface I

We consider a vector class that stores **intList::LinkedList** objects.

```
namespace vecIntList {

    class vector{
    private:
        std::allocator<intList::LinkedList> alloc; // allocator to create/destroy
        size_t sz, cap; // size and capacity
        intList::LinkedList* listptr; // pointer to beginning of memory segment

        friend void swap(vector&, vector&); // to swap two vectors
```

Vector Interface II

```
public:
    // iterators will just be pointers!
    using iterator = intList::LinkedList*;
    using const_iterator = const intList::LinkedList*;
```

Vector Interface III

```
    vector(); // default constructor
    vector(const vector&); // copy constructor
    vector(vector&&); // move constructor
    vector& operator=(vector) &; // assignment operators!
    ~vector(); // destructor

    // to run on const or non-const vectors
    const intList::LinkedList& operator[](size_t) const;
    intList::LinkedList& operator[](size_t);

    // begin and end, overloaded on const
    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;
};
```

Vector Interface IV

```
    void swap(vector&,vector&);

} // closing namespace brace
```

Vector Interface

Rather than writing our own iterator classes, we will sneakily use "iterator" and "const_iterator" to refer to raw pointers. After all, pointers already have the pointer arithmetic, dereferencing, and comparison functions we need!

We overload the subscript **operator[](size_t)** on const, along with the **begin**/**end** member functions.

The **iterator** (ordinary pointer) will allow modification of the vector elements but the **const_iterator** (pointer to const) will not.

Vector Constructors

For implementations, we assume work is done within the **vecIntList** namespace.

```
vector::vector() : cap(1), sz(0), listptr( nullptr ) {
   // if fails then memory not allocated
   listptr = alloc.allocate(cap);
}

vector::vector(vector&& rhs) : vector() {
   swap(*this, rhs);
}
```

Vector Constructors

```
// make initial capacity twice the initial size
vector::vector(size_t size) : sz(size), cap(2*size), listptr( nullptr) {
    listptr = alloc.allocate(cap); // if fails, memory not allocated

    for (size_t i=0; i < sz; ++i) { // loop over desired size
        try { // try to make each object
            alloc.construct(listptr+i); // default construct elements
        }
        catch(...) { // but if construction fails
            for (size_t j=0; j < i; ++j) { // over all object made (ith was not made)
                alloc.destroy(listptr + (i-j)); // destroy them
            }
            alloc.deallocate(listpr, cap); // deallocate

            throw; // and throw again
        }
    }
}
```

Vector Constructors

There is a danger that the allocator could throw an exception during the allocation or construction phases.

If the **allocate** fails then an exception would be thrown but the **listptr** would still point to **nullptr** not managing heap memory.

If the **construct** fails, some objects could have already been made and we need to destroy all those objects and free the memory before **throw**ing the exception onwards.

Vector Operations

```
vector::~vector(){
   if(sz > 0) { // if anything was actually constructed
      for (auto p = listptr + sz - 1; p != listptr; - -p) { // from end to begin
         alloc.destroy(p);
      }
   alloc.destroy(listptr); // and destroy first element
   }

   alloc.deallocate(listptr, cap); // free the block of memory
}

void swap(vector& left, vector& right){
   std::swap(lhs.listptr, rhs.listptr);
   std::swap(lhs.cap, rhs.cap);
   std::swap(lhs.sz, rhs.sz);
}
```

Vector Push Back - Efficiently

Suppose that during a **push_back** operation, the size was being pushed beyond the vector capacity and a new segment of memory needs to be allocated.

Suppose we store a copy of **listptr** as **tempptr** and a copy of **cap** as **oldcap** before doubling **cap** and then transfer the old data as below:

```
// ...
listptr = alloc.allocate(cap); // reallocate to the double capacity

for (size_t i = 0; i < oldcap; ++i) { // MOVE over values
   alloc.construct(listptr+i, std::move(*(tempptr + i) ) );
}
for (size_t i=0; i < oldcap; ++i) { // destroy old values
   alloc.destroy(tempptr + oldcap - 1 - i);
}

alloc.deallocate(tempptr, oldcap); // free memory block
```

Vector Push Back - Efficiently

Note the call to **std::move**: without it, the allocator would copy construct each object in the new space because ***(tempptr+i)** is an lvalue. This would be inefficient.

For brevity, **try** and **catch** have not been included here, but they should be used in case the construction fails.

Vector Costing

Given an index, the retrieval time for an element in a vector is $O(1)$. Vectors support **random access**.

Appending an item to the end is an $O(1)^+$ process, sometimes called "amortized O(1)". Most of the time takes $O(1)$ steps to append, but every so often new space needs to be allocated and elements moved, costing $O(n)$. These additional costs are rare enough that it averages out to $O(1)$. Removing an item at the end is $O(1)$.

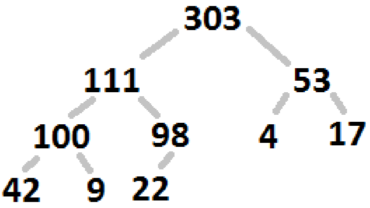
Insertion/deletion at an arbitrary point is $O(n)$ because as many as all n elements may need to be shifted.

Maximum Heap Idea

A **maximum heap** is a binary tree structure such that

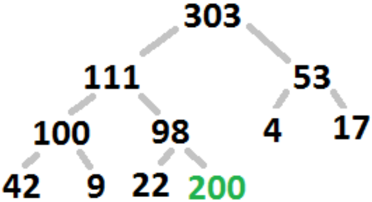
- each node is larger than or equal to its child nodes and
- the structure is **almost complete**: it is filled from left to right, top to bottom, with the top layer storing one value, the second layer storing 2 values, the third layer storing 4 values, etc. Each node has two child nodes except for possibly those in the last level.

Its main purpose is to keep the biggest (highest priority) item at the top for fast access.



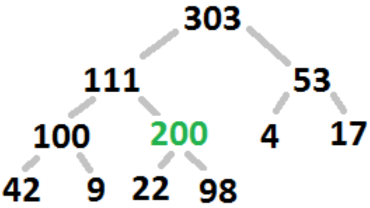
Maximum Heap Idea

To insert an item, it is first placed in the newest possible position. Then, it is compared against its parent and it swaps with its parent if its value is larger. The process stops when it is found to be smaller than its parent or at the top of the heap.



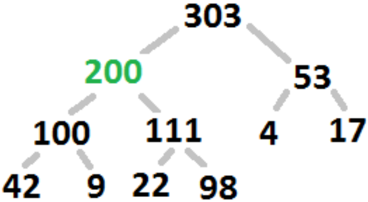
Maximum Heap Idea

To insert an item, it is first placed in the newest possible position. Then, it is compared against its parent and it swaps with its parent if its value is larger. The process stops when it is found to be smaller than its parent or at the top of the heap.



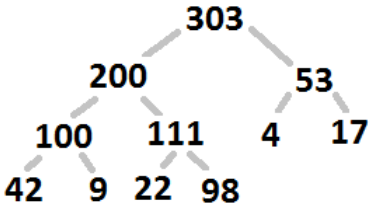
Maximum Heap Idea

To insert an item, it is first placed in the newest possible position. Then, it is compared against its parent and it swaps with its parent if its value is larger. The process stops when it is found to be smaller than its parent or at the top of the heap.



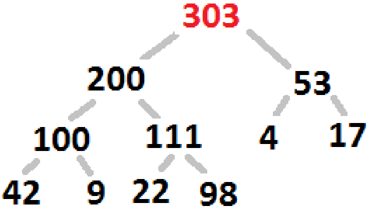
Maximum Heap Idea

To insert an item, it is first placed in the newest possible position. Then, it is compared against its parent and it swaps with its parent if its value is larger. The process stops when it is found to be smaller than its parent or at the top of the heap.



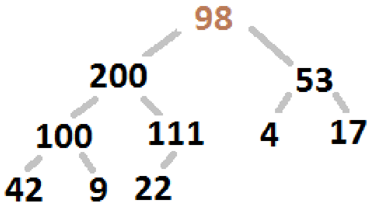
Maximum Heap Idea

To remove the top item, the item at the final position is moved to the top. It is then demoted and swaps with its largest child and so on, until no more demotions are necessary, i.e., the item is at the bottom or it is at least as large as both its children.



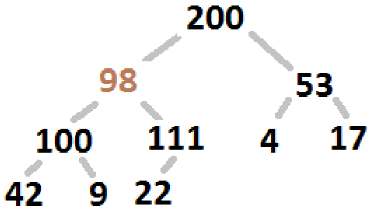
Maximum Heap Idea

To remove the top item, the item at the final position is moved to the top. It is then demoted and swaps with its largest child and so on, until no more demotions are necessary, i.e., the item is at the bottom or it is at least as large as both its children.



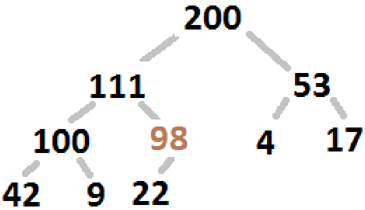
Maximum Heap Idea

To remove the top item, the item at the final position is moved to the top. It is then demoted and swaps with its largest child and so on, until no more demotions are necessary, i.e., the item is at the bottom or it is at least as large as both its children.



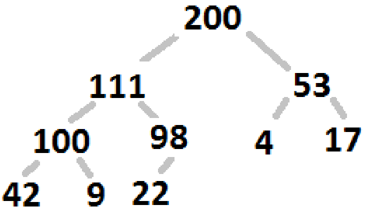
Maximum Heap Idea

To remove the top item, the item at the final position is moved to the top. It is then demoted and swaps with its largest child and so on, until no more demotions are necessary, i.e., the item is at the bottom or it is at least as large as both its children.



Maximum Heap Idea

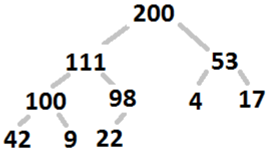
To remove the top item, the item at the final position is moved to the top. It is then demoted and swaps with its largest child and so on, until no more demotions are necessary, i.e., the item is at the bottom or it is at least as large as both its children.



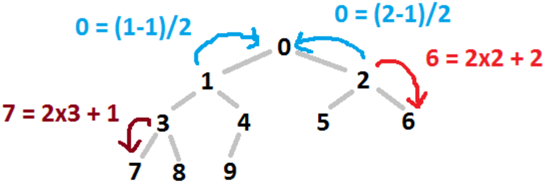
Maximum Heap Idea

The heap can be easily managed internally as a **std::vector** because its regular shape makes for a simple relationship between each node and its parent nodes. Consider indexing the heap from 0. Then *with integer division*:

- Parent_Index = (Child_Index - 1)/2;
- Left_Child_Index = 2 $\times$ Parent_Index + 1;
- Right_Child_Index = 2 $\times$ Parend_Index + 2.



Visualization



Vector Indices

200	111	53	100	98	4	17	42	9	22

Vector Storage

Maximum Heap Idea

Both insertion and root removal are $O(\log n)$ processes for a maximum heap, even if the data come in sorted (unlike for a binary search tree where this could degenerate to $O(n)$). On the other hand, data in max heaps are not sorted besides for having the largest item at the top.

Hash Table Motivation

In an ideal world, we could manage very large sets of data and search/modify/insert/delete with a time cost of $O(1)$, independent of the amount of data we are already storing.

In a truly ideal world, we could imagine that we have a large block of contiguous memory for random access. Each block of memory could be empty (somehow) or could store a single value. If a new value were to come in, it could be given its own designated spot, unique for that precise value. Then, everything would be $O(1)$...

Hash Table Motivation

Supposing the existence of some function that maps an input to an index of a very large vector with *k* spots, we ponder the question: *is it realistic that every piece of data can have its own spot? And if not, how many pieces of data n does it take before two or more data would need to share an index-position of a vector, a "collision"?*

Birthday Problem

A more fun variant: think of the number of days in the year as $k = 365$, and we would like to know how many people in the room n it takes before two or more people will share the same birthday. We'll assume all birthdays are equally likely (even though that's not true).

In a room of n people, the probability p that 2 or more share the same birthday is:

$$\begin{aligned} p &= 1 - \Pr(\text{all birthdays are different}) \\ &= 1 - \frac{\text{all permissible distinct birthday combinations}}{\text{all possible birthdays}} \\ &= 1 - \frac{365 \times 364 \times 363 \times (365 - (n-1))}{365^n} \\ &\underbrace{\approx}_{\text{mathemagic}} 1 - e^{-n^2/(2k)} \end{aligned}$$

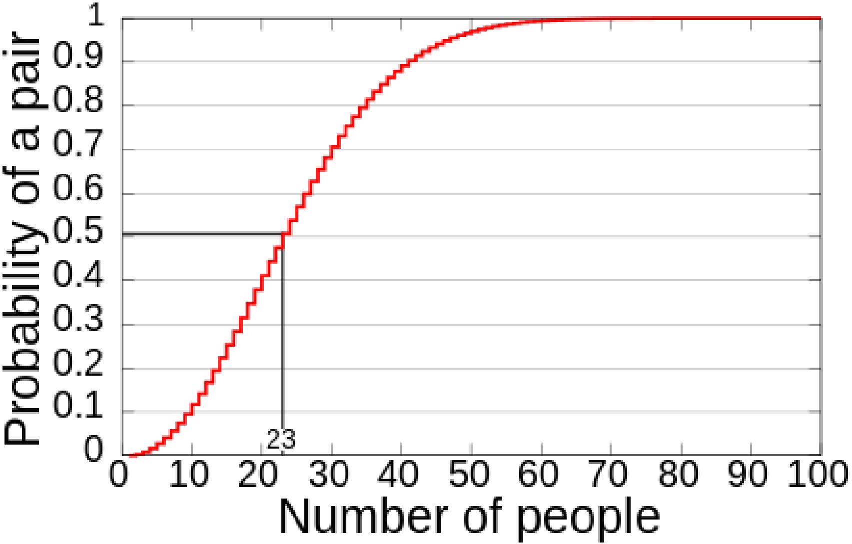
provided number of people n is much less than k = 365.

Birthday Problem

Moral:

- If $n \approx 0$, $p \approx 0$ because $e^{-n^2/(2\times 365)} \approx 1$.
- Once $n \approx O(\sqrt{k}) = \sqrt{365} \approx 20$, p is non-negligible and rapidly increases because $e^{-n^2/(2\times 365)}$ isn't so close to 1 anymore and decreases quite quickly with n.
- Also $p = 1$ if $n > 365$ since there are more people than possible days in the year so there would have to be a collision.

Birthday Problem



(from *https://en.wikipedia.org/wiki/Birthday_problem#/media/File:Birthday_Paradox.svg*)

General Collisions and Hash Maps

So how bad is it for data storage?

If we tried to give every piece of data its own vector element, once the number of data members we wish to store n is on the order of $\sqrt{k}$, where k is the size of the vector, we will likely have collisions.

Even a vector with a million components would likely have data collisions with as few as ≈ 1000 items.

General Collisions and Hash Maps

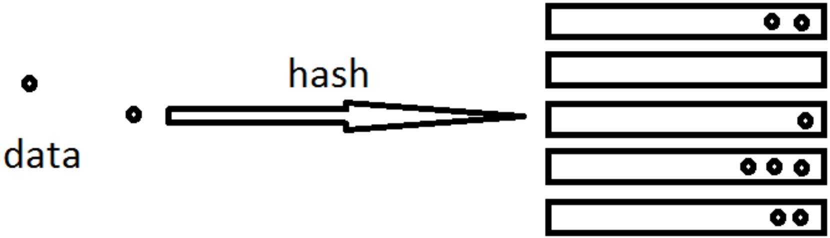
These problems can be mitigated with a **hash map**.

A hash map (or **hash table**) is a data structure such as a vector of "buckets" to store data, where each bucket is a binary search tree, linked list, another vector, etc.

It also includes a **hashing** function to assign each element an index of the vector, i.e., which bucket it belongs to.

Hash Map Outline

- Create a vector to store lists or sets or other vectors.
- Insert: using a hashing function, map data to the vector component ("bucket") where it belongs and insert into list/set/vector.
- Find/remove: find the appropriate bucket and search/remove from the corresponding list/set/vector.



Hash Map Implementation for Storing Integers I

Our **vecIntList::vector** class that stores **listInt::LinkedList** objects is pretty handy!

```
class hashMap {
private:
    size_t size; // desired size
    vecIntList::vector vec; // vector of LinkedLists storing ints

    // obtain index from 0 to size-1
    size_t hash_it(int i) const {
        size_t pos = static_cast<size_t>( (i>0) ? i : (-i)); // get positive value
        return pos % size;
    }

public:
    hashMap(size_t s) : size(s), vec(s) {} // constructor
```

Hash Map Implementation for Storing Integers II

```
void insert(int i) { // add value to hash map
    size_t index = hash_it(i); // find vector index where i belongs

    auto itr = vec[index].begin(), end = vec[index].end(); // find range

    // look through the entire list at that index
    for( ; itr != end; ++itr) {

        if ( *itr == i ) { // if there is a match
            break; // break out, because there is a duplicate
        }
    }

    if(itr == end ) { // if at end then not found so
        vec[index].push_back(i); // add to list
    }
}
```

Hash Map Implementation for Storing Integers III

```
    void remove(int i); // removes value if found

    bool contains(int i) const; // checks if i found
};
```

Costing of Hash Maps

A good hashing function should carefully distribute the data so that no single vector element shoulders all the burdens.

Assuming a good function has been written, then the **load factor** $L = n/k$ should be small where n is the number of data being stored and k is the number of buckets in the vector.

L represents the average number of data there are in any given vector bucket.

Costing of Hash Maps

Then for a hash table implemented as a vector of linked lists, access/insertion time is $O(1) + O(L) \approx O(1)$ since L shouldn't really grow.

For a hash table implemented as a vector of binary search trees, access/insertion time is $O(1) + O(\log L) \approx O(1)$ since L shouldn't really grow.

This breaks down when L grows large, but even then, the $O(\log L)$ term doesn't grow too fast when binary search trees are stored in the buckets (better than lists, but does require **operator<** or some comparison function).

Aside: std::forward_list

std::forward_list is found in the **<forward_list>** header.

A **std::forward_list** only allows insertions at the front and only allows forward iterators: one cannot go backwards in the container.

Summary

- Most data structures are implemented based upon a vector (allocating data contiguously); a linked list (storing data in nodes that reference their previous/next node); or a binary search tree (storing data in a sorted order in nodes that reference child nodes and/or parent nodes).
- Iterators are wrappers for pointers and have well-defined behaviours within a containing class.
- Diagrams are helpful in studying data structures and their implementations!
- Linked lists offer $O(1)$ insertion/deletion given a position and $O(n)$ lookup.
- Binary search trees offer $O(\log n)$ insertion/deletion and $O(\log n)$ lookup.
- Vectors offer $O(n)$ insertion/deletion given a position and $O(n)$ lookup.

Summary

- Classes can be **nested** for hiding implementations and tying classes together.
- Allocators set aside unconstructed memory, and can create and destroy that memory.
- Hash tables are handy for large collections of data.
- The standard includes many templated classes including **array**, **vector**, **deque**, **priority_queue**, **stack**, **queue**, **list**, **forward_list**, **pair**, and **initializer_list**; also there are ordered and unordered (prepend **unordered_** to name) versions of **set**, **multiset**, **map**, and **multimap**.
- The unordered versions are implemented as hash maps.

Exercises I

1. Implement a forward list data structure such that elements can only be inserted at the front, and traversal can only go from front to back.
2. Write a **"multiset"** data structure by extending the binary search tree idea to allow for multiple copies of an element to be stored. Don't worry about self-balancing like the real **std::multiset**.
3. Write a **"map"** data structure by extending the binary search tree idea to allow for key-value pairs to be stored, sorted based on the keys. Don't worry about self-balancing like the real **std::map**.
4. Give the full implementation of a **vector** class that uses allocators.
5. Write a **hash table** to store **std::string**s. You can use a **std::vector**$<$**std::vector**$<$**std::string**$>>$ as an underlying container. For a hash function, try:
    - one that is just the size of the string mod the number of buckets; and
    - one that is the sum of the ASCII values mod the number of buckets.

   Be sure to define a **default constructor**, plus **insert**, **erase**, and **find** functions.

Exercises II

6. How is recursion useful for a binary search tree in:
   - inserting an item;
   - copying a tree so the new tree has the identical structure;
   - destructing a tree
7. Determine, giving an argument/justification, the runtime cost of:
   - copying a binary search tree recursively;
   - destructing a binary search tree recursively;
   - traversing a binary search tree (this one is tricky so here's the answer: at best it is $O(n)$; at worst it is $O(n \log n)$);
   - copying a binary search tree from beginning to end, without recursion, without preserving structure;
   - destructing a binary search tree from beginning to end, without recursion.
8. Implement a **maximum heap** for **int**s. Be sure to include:
   - a **default constructor**;
   - an **insert** function;
   - a **pop** function; and
   - a **top** function, returning what's on top.

Exercises III

9. Compare and contrast memory management via the **new** and **delete** expressions and via **std::allocator**s.
10. What is **Argument Dependent Lookup**, in what contexts does it arise, and how is it helpful?
11. Consider two possible variants of implementing a **LinkedList** class (who cares what data type it stores). In the first, **node**, **iterator**, and **const_iterator** are classes that exist outside of the **LinkedList** scope; in the second variant, they are nested within **LinkedList**.
    11.1 Give a rationale for why the C++ Standard adopts containers that follow the second implementation, rather than the first.
    11.2 Assuming the classes are well-encapsulated with private constructors, etc., for both variants, identify which classes need to declare which other classes as **friend**s.

Exercises IV

12. For each scenario below, identify the most appropriate data structure(s) within the C++ Standard Library:
    - You need to go through a large text file storing a list of words. You must determine the number of times each individual word appeared.
    - You need to store a large collection of **Simulation** objects (objects that store properties relevant to some simulations - what they store isn't all that important). You need to frequently traverse the collection from beginning to the end and remove objects during the traversal; you also need to periodically add more **Simulation** objects to the collection.
    - You need to store exactly **100** integers in a data structure. You will not need to remove any but you will need to access elements at random.
13. Consider the implementation for either the **Linked List** or **Binary Search Tree.** Notably, raw pointers were used.
    - Why were pointers used rather than nodes storing references or copies of other nodes?
    - Could smart pointers offer a viable alternative to raw pointers? Discuss when they would/would not work and the possible tradeoffs.