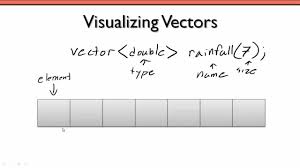
Vector:

|  |  |
| --- | --- |
| Initialize | std::vector<int> vname; |
| Insert | vname.insert(itr\_i, val) |
| Erase | Vname.erase(itr\_position)  Moves everything behind the value up one spot |
| Push\_back | Vname.push\_back(val)  Sometimes allocated a new array of size 2\*m\_alloc |
| Pop\_back | Vname.pop\_back()  Does not return a value |
| Iterators | Vector<int>::iterator v\_itr = v.begin()  Has operator[]  Uses operator<  vector<T>::insert() returns an itr b/c insert() may invalidate the itr passed in  When iterating: returns iterator to same place in list, but really the next element:   * name.insert(itr—) * name.erase(itr--) |



|  |  |
| --- | --- |
| **Time Complexities** | |
| Size | O(1) |
| push\_back: | O(1) |
| erase: | O(n) |
| insert: O(n) | O(n) |
| pop\_back: | O(1) |
| resize: | O(n) |
| operator=: | O(n) |
| Clear | O(n) |
| sort | O(nlog(n)) |

|  |
| --- |
| * uses an array as the underlying representation |

Reference: <http://www.cplusplus.com/reference/vector/vector/>

Lab 5

|  |  |
| --- | --- |
| **Vector vs Array** | |
| Knows how many elements it contains |  |
| Can store elements of any type | |
| **NEITHER** Prevents access of memory beyond its bounds | |
| Is dynamically resizable |  |
| Can be passed by reference | |

Array Reference: <http://www.cplusplus.com/reference/array/array/>

STL List/ DSList

* Implemented as a doubly linked list
* Performs better in insert, erase and moving elements (compared to array/vector)
* Lack operator[ ], takes more memory to store pointers to next elements

|  |  |
| --- | --- |
| Initialize: | std::list<int> Lname |
| Insert: | Lname.insert(itr\_i, val) |
| Erase: | Lname.erase(itr\_position) |
| Push\_back | lname.Push\_back(value)  Sometimes alloc new array size 2\*m\_alloc |
| Pop\_back | Lname.pop\_back()  Does not return a value |
| Iterators | list<int>::iterator l\_itr = l.begin()  No operator[]  Uses operator<  Incrementing the end() iterator in any STL list has undefined behavior  When iterating: returns iterator to same place in list, but really the next element:   * name.insert(itr—)   name.erase(itr--) |

|  |  |
| --- | --- |
| Time Complexitites | **Test Review** |
| Size: O(1)  push\_back/push\_front: O(1)  erase: O(1)  insert: O(1)  pop\_back/pop\_front: O(1)  resize: O(n)  operator=: O(n)  clear: O(n)  sort: O(nlog(n)) |  |

Moving through a List with pointers

**A close up of text on a black background

Description automatically generated**

A picture containing skiing, air, jumping, riding

Description automatically generated

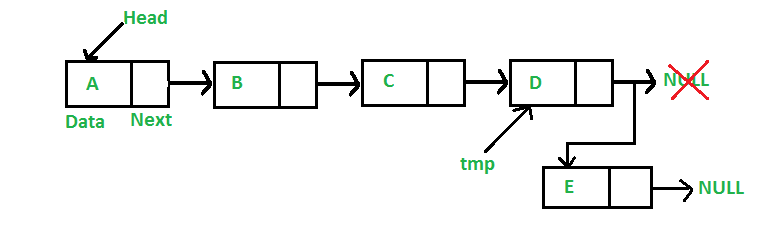
Reference: <http://www.cplusplus.com/reference/list/list/>

Lab 7

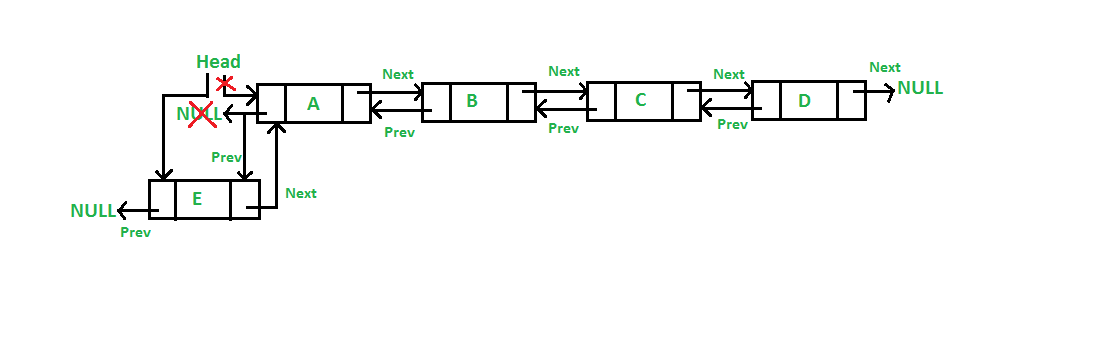
|  |
| --- |
| * allows efficient (sublinear) removal of the first and last elements (or the minimum and maximum elements) * uses a network of nodes connected by pointers as the underlying representation * allows sublinear merging of two of instances of this data structure |

Singly Linked List

|  |  |
| --- | --- |
| Initialize: | Node<T> \*head = NULL; |
| Pop\_back/push\_back/Insert/Erase -> First iterate to the index you want to remove (minus one index | |
| Insert: | Insert B into T:  B->next= T->next;  T->next = B; |
| Erase: | P(rev) = T(curr);  T = T->next  P->next= T->next  delete temp |
| Push\_back | P->next = T;  T->next = NULL |
| Pop\_back | P->next = T;  T->next = NULL |
| Copy | Start from the back, see Question 5 of review |



|  |
| --- |
| Time Complexity |
| Size: O(n)  push\_back: O(n)  erase: O(1)  insert: O(1)  pop\_back: O(n) |

Doubly-Linked List:

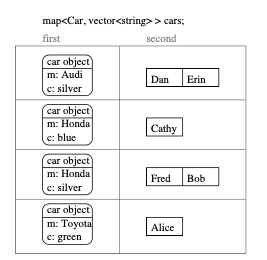
|  |  |
| --- | --- |
| Initialize: | Node<T> \*head = NULL; |
| Pop\_back/push\_back/Insert/Erase -> First iterate to the index you want to remove (minus one index | |
| Insert: | Insert B into T:  B->prev = T;  T=t->next;  B->prev->next = B;  T->prev = B;  T->prev->next = T; |
| Erase: | P(rev) = T(curr);  T = T->next;  P->next = T->next  P->next->prev=P  delete T |
| Push\_back | P->next = T;  P->next->prev = P  T->next = NULL; |
| Pop\_back | P = T  T = T->next  P->next->prev = NULL  P->next = NULL |

HW 5

Maps:

* **Associative** - referenced by their *key* and not by their position
* **Ordered (operator<) –** Everything follows a strict order
* **Unique Key – No Duplicates**

|  |  |  |
| --- | --- | --- |
| Initialize: | Node<T> \*head = NULL; |  |
| Erase | Mymap.erase(“A”); //Erase by key  Mymap.erase(itr); //Erase by iterator  Mymap.erase(itr, mymap.end()) //Erase by range  To Erase a part of a map, use iterators and .find | O(log n)  O(1)  O(n) |
| Find | Itr = mymap.find(“b”);  Returns map.end() if not found | O(log n) |
| Insert | Mymap.insert(make\_pair(“A”, 12180);  Pair: pair<string, int> p; | O(log n) |
| Operator [ ] | Mymap[“A”] = 12830  If does not match another key, inserts the element  If does match another key, returns reference to mapped value | O(log n) |
| Iterators | Map::<string, int>::iterator itr = map.begin();  Has “first” and “second” | |
| Printing | Use Iterators so that you can print alphabetically | |

A screenshot of a map

Description automatically generated

Reference: <http://www.cplusplus.com/reference/map/map/map/>

Lab 9

|  |
| --- |
| * allows efficient (sublinear) removal of the first and last elements (or the minimum and maximum elements) * uses a network of nodes connected by pointers as the underlying representation * entries cannot be modified after they are inserted (requires re-insertion or re-processing of position) |

Queues:

**Queues** – pushed into the BACK and removed from the FRONT (First In First Out)

* Cannot access middle
* Implement using a LIST. Can use vectors but requires more work.
* All Queue Operations are O(1), using a list

Copy Constructor: queue<double> temp(q);

|  |  |
| --- | --- |
| Initialize: | Queue<double> q; |
| Pop | Myqueue.pop();  Removes myqueue.front();  Does not return a value |
| Push | Myqueue.push(12.45); |
| Other helpful queue aspects | |
| Front | Double d = myqueue.front(); |
| Back | Double d = myqueue.back();  Myqueue.back() -= myqueue.front(); |
| Size | Myqueue.size() |

A screenshot of a video game

Description automatically generated

Refer to list for more information

Reference: <http://www.cplusplus.com/reference/queue/queue/>

Priority Queues:

**Priority Queue** – Used to prioritize operations (to-do list, events in a simulation)

* Has a *front* and *back*
* Each item is stored using an associated “priority”
  + The top item has the lowest value priority score. The back is never accessed through the public interface.

**Implementing a Priority Queue**

* List (or Vector) sorted or unsorted
  + At least one of Push/Pop will cost O(1) time.
* Binary Search Trees
  + Priority is the **key:**
    - Pop – combination of finding the minimum key and erase
    - Push – Ordinary BST insert
  + O(log n) – even with balancing

Reference:

Lab 12

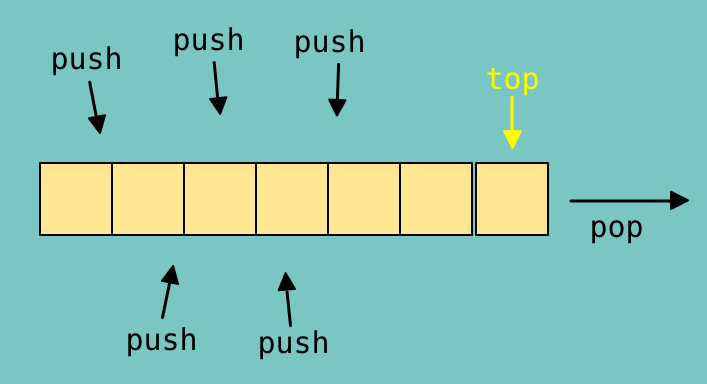
Lectures

|  |
| --- |
| * Uses array or vector as underlying representation * The underlying data structure must be “balanced” or well- distributed to achieve the targeted performance (if BST) * requires definition of operator< or operator> * entries cannot be modified after they are inserted (requires re-insertion or re-processing of position) |

Height: log(n)

Bottom row starts

A picture containing bird

Description automatically generated

Heaps:

* Implementation of priority queues.
* Heap Sort:
  + Make it into a Max Heap with Heapify
  + Then pop off the top node each time and

Reference: Lab 12, Lecture 21

**Percolate UP/Down**A screenshot of a social media post

Description automatically generated

Memory and Pointers

Anytime you write a class that uses ***Dynamically Allocated Memory*** you need these **4** essential functions**:**

* **Default Constructor** Office();
* **Destructor** ~Office();
* **Copy Constructor** Office(const Office& office);
* **Assignment Operator** Office& operator=(const Office &office);

\*All of these can go anywhere in the public interface.

Miscellaneous:

* All looping constructs are equivalent and any algorithm written using one looping construct can be re-written with the others. **For** == **While**

Garbage Collection:

|  |  |
| --- | --- |
| Explicit Memory Management (C++) | High memory usage, fragmentation of data is minimized (Hand-Held Game, PS4) |
| Reference Counting | Tree based systems (Chess) |
| Stop & Copy | Very high-performance speed with very little memory usage. (Student Registration System)  Forwarding Address - When a non-garbage cell is copied from the old memory partition to the new memory partition, a forwarding address is left to indicate that the cell has been copied and to provide the address of the new location. All references to the old location will see the forwarding address and can then be updated as appropriate. If we don’t record this forwarding address cyclical data structures will not be copied correctly: the garbage collector will repeatedly copy the same cell resulting in an infinite loop! |
| Mark-Sweep | low memory Usage, slow speeds (Webserver) |

Operators:

**3 Types:**

* Non-member function
  + Only if it does not require access to any of the private variables stored in the class
* Member function
  + If it does require access to private variables (or if the first argument is that class), only used for that class
* Friend function
  + It requires access to private member variables and can by used without reference to that class

**Operator between two classes:**

Superhero& operator/=(const string &id);

return \*this to ensure that you are editing the class.

**Operator< for Alphabetically:**

A picture containing drawing

Description automatically generated

Review:

How to declare the head of a new linked list. Syntax

How list iterators work, do you need to deference them? (is \* needed) Declaring iterators to stuff on the heap vs stuff on the stack.

How are lists structured? How to do access the head and tail given “Node<T> & lst”

Types and operations on types (16)