FINAL EXAM REVIEW

Vector:

Initialize	`std::vector<int> vname;`
Insert	`vname.insert(itr_i, val)`

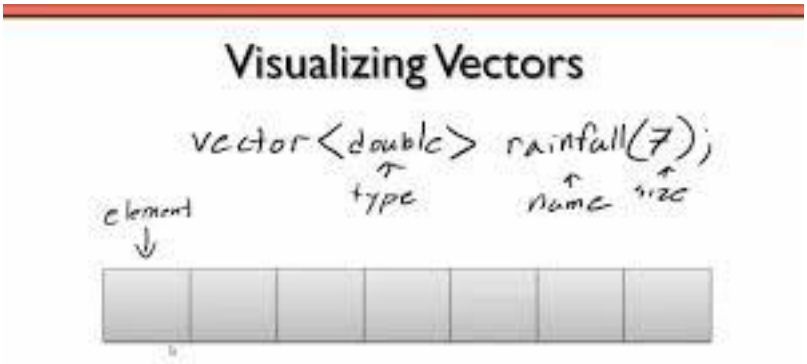
Erase	`Vname.erase(itr_position)`
	
	`Moves everything behind the value up one spot`

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/

FINAL EXAM REVIEW

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)`

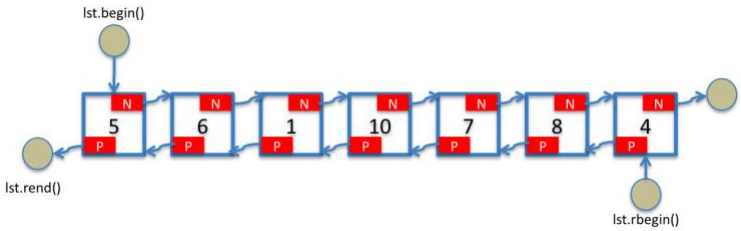
Push_back	`lname.Push_back(value)`
	
	`Sometimes alloc new array size 2*m_alloc`

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




```
template <class T>
void dslist<T>::reverse(){
    // Handle empty or single node list
    if (head_ == tail_) return;
    // Swap pointers at each node of the list,
    //using a temporary // pointer q to remember where to
    Node<T>* p = head_;
    while (p) {
        Node<T>*q = p->next_;
        p->next_ = p->prev_;
        p->prev_ = q;
        p = q;
    }

    p = head_;
    head_ = tail_;
    tail_ = p;
```

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

FINAL EXAM REVIEW

Singly Linked List

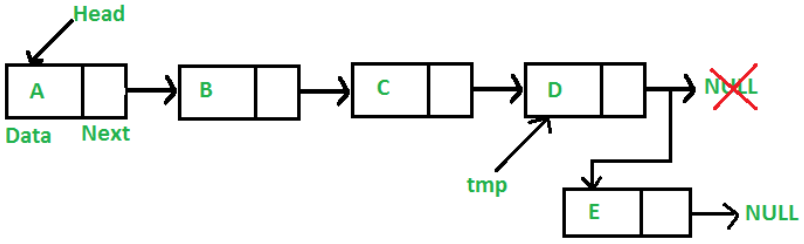
Initialize:	`Node<T> *head = NULL;`
`Pop_back/push_back/Insert/Erase -> First iterate to the index you want to remove (minus one index`	

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

```
Node<T>* new_head = new Node<T>;
```

FINAL EXAM REVIEW

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`	

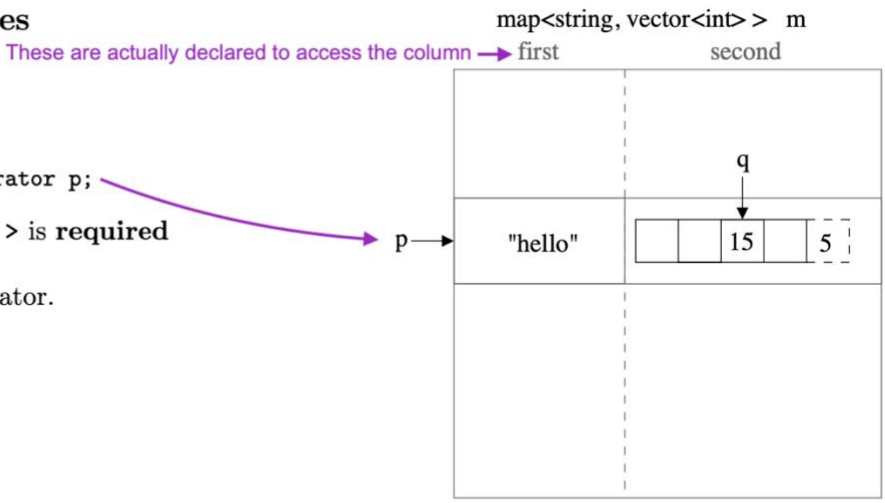
15.1 More Complicated Values

- Let's look at the example:

```
map<string, vector<int> > m;
map<string, vector<int> >::iterator p;
```

Note that the space between the > > is **required** (by many compiler parsers).
Otherwise, >> is treated as an operator.

This is kind of long, USE A TYPEDEF
- easier if there is a mistake

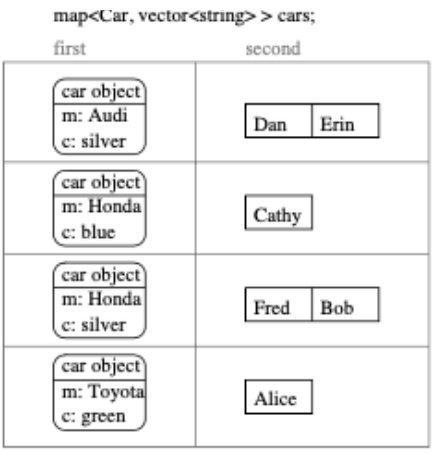


Maps sort by key, so make sure your diagram is sorted by key operator<

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)



FINAL EXAM REVIEW

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;`

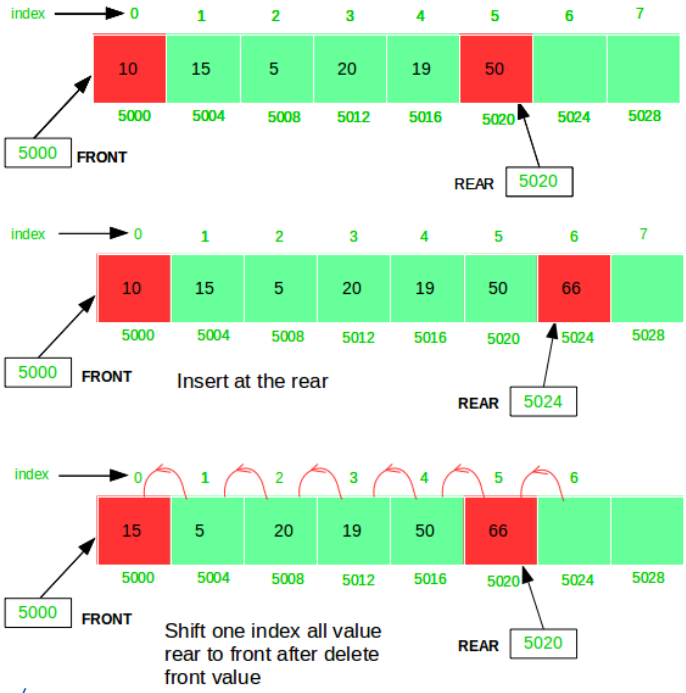
Pop	`Myqueue.pop();`
	
	`Removes myqueue.front();`
	`Does not return a value`

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()`

Refer to list for more information



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

FINAL EXAM REVIEW

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

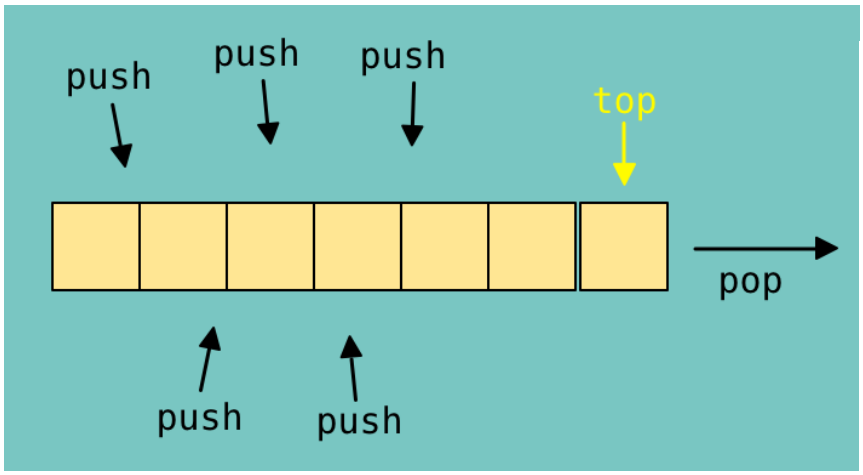
- List (or Vector) sorted or unsorted
    - At least one of Push/Pop will cost O(1) time.
- Binary Search Trees
    - Priority is the **key:**
        - Pop – combination of finding the minimum key and erase
        - Push – Ordinary BST insert
    - O(log n) – even with balancing

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

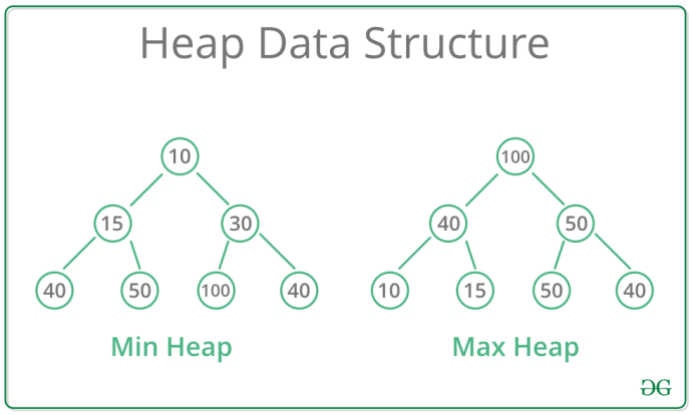



```
void pop_max() {
  assert(!m_heap.empty());
  int tmp = (size()+1)/2;
  for (int i = tmp+1; i < size(); i++) {
    if (m_heap[tmp] < m_heap[i])
      tmp = i;
  }
  m_heap[tmp] = m_heap.back();
  m_heap.pop_back();
  this->percolate_up(tmp);
}
```

FINAL EXAM REVIEW

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

21.1 Implementing Pop (a.k.a. Delete Min)

- The value at the top (root) of the tree is replaced by the value stored in the last leaf node.
  This has echoes of the erase function in binary search trees.
- The last leaf node is removed.
  QUESTION: But how do we find the last leaf? Ignore this for now...
- The value now at the root likely breaks the heap property. We use the `percolate_down` function to restore the heap property. This function is written here in terms of tree nodes with child pointers (and the priority stored as a `value`), but later it will be written in terms of vector subscripts.

```
percolate_down(TreeNode<T> * p) {
  while (p->left) {
    TreeNode<T>* child;
    //  Choose the child to compare against
    if (p->right && p->right->value < p->left->value)
      child = p->right;
    else
      child = p->left;
    if (child->value < p->value) {
      swap(child, p); // value and other non-pointer member vars
      p = child;
    }
    else
      break;
  }
}
```

```
While we can keep going left:  //The first child is alwasys on the left
    If right, and right<left{
        this is your child
    else do the same with the left
```

21.2 Implementing Push (a.k.a. Insert)

- To add a value to the heap, a new last leaf node in the tree is created to store that value.
- Then the `percolate_up` function is run. It assumes each node has a pointer to its parent.

```
percolate_up(TreeNode<T> * p) {
  while (p->parent)
    if (p->value < p->parent->value) {
      swap(p, parent);  // value and other non-pointer member vars
      p = p->parent;
    }
    else
      break;
}
```

```
While we can move up:
    if my value is smaller than my parents value
        swap me with my parent
```

FINAL EXAM REVIEW

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)

FINAL EXAM REVIEW

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:

```
bool operator<(const Car &a, const Car &b){
    if(a.getmaker() == b.getMaker()){
        return a.getColor() < b.getColor();
    }else{
        return a.getMaker() < b.getMaker();
    }
}
```

FINAL EXAM REVIEW

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)