**Queues**

* **Queue** – first-in-first-out
  + **initQueue** – creates & returns a new empty queue
    - Pre-condition – nothing
    - Post-condition – a new empty queue
  + **enter** (enqueue) – takes new element and queue and returns e1, … en, enew
    - Pre-condition – a queue
    - Post-condition – a queue with new element added at the end
  + **leave** (dequeue) – takes queue and returns e2, … en
    - Pre-condition – a non-empty queue
    - Post-condition – a queue with first element removed
  + **first** (front) – takes queue and returns first element e1
    - Pre-condition – a (non-empty?) queue
    - Post-condition – the same queue
  + **isEmpty** – takes queue and returns if n == 0
    - Pre-condition – a queue
    - Post-condition – the same queue
  + **nuke**
    - Pre-condition – a queue
    - Post-condition – nothing (nullptr)
* Vector as a queue
  + enter(e) calls push\_back(e)
  + Must keep track of the first element as well
  + A linear array is space-inefficient – enqueuing and dequeuing will shift the queue forward in the array, leaving empty memory
* Linked list as a queue
  + First node points to next element as well as last element
* **Defensive programming**
  + It’s easy to make mistakes with pointers
  + Two approaches:
    - Use a debugger
    - Inserting print statements & assertions
  + When to insert a print statement?
    - At the beginning of a function call
    - At the end of a function call
    - Before deleting something
    - Print the whole data structure (for debugging only)
  + When to use asserts?
    - Check our logic/assumptions are correct
    - Check the user is being reasonable (e.g. popping an empty stack)
* **Dynamic arrays**
  + Usually use STL vector instead
* C++ arrays – declared statically
  + Array bound must be a compile-time constant (const int N → a[N])
* Or declared dynamically
  + Storage located on the heap
  + E.g. int\* a = new int[N]; … delete [] a;
  + Extent of the array is stored at the beginning (a[-1])
* **Variants of the linked list**
* Doubly linked list – can traverse forwards or backwards
  + Pointers to previous and next nodes
* Binary tree
  + Pointers to left & right nodes
* Getting linked structures right
  + Draw pictures
  + Cases:
    - Empty list, one element list, etc.
    - First element, last element, duplicate elements etc.
  + Testing
    - Write a file with expected output, then run each test and compare the outputs with diff
    - Regression testing – retest everything when changes are made
    - Test-driven development – develop test cases that satisfy the requirements before writing the code
* FIFO, LIFO etc. – ordered lists (ordered by insertion)
* **Sorted list** – ordered by key value
  + Looking for an item in a sorted linked list

bool lookup (const SortedList & first, string val) {

Node \* p = first;

While (nullptr != p && p->val < val) { //short-circuit eval

p = p->next;

}

return (nullptr != p && p->val == val);

}

* + Inserting an item

Void insert (SortedList & first, string val) {

Node \* newNode = new Node;

newNode->val = val;

if (isEmpty(first) || val <= first->val) { //is first element

newNode->next = first;

first = newNode;

}

else { //is after the first element

Node \* p = first;

While (nullptr != p->next && val > p->next->val) {

p = p->next;

}

newNode->next = p->next;

p->next = newNode;

}

}

* + Removing an item

Void remove (SortedList & first, string val) {

assert(!isEmpty(first));

Node \*temp;

if (first->val == val) { //is first element

temp = first;

first = first->next;

}

else { //is not first element

Node \*cur = first;

while(cur->next != nullptr && cur->next->val < val) {

cur = cur->next;

}

if (cur->next == nullptr || cur->next->val != val) {

cerr << “not found” << endl;

return;

}

temp = cur->next;

cur->next = cur->next->next;

}

delete temp;

}

* Pre- & post-conditions for sorted linked list:
  + Enter:
    - Pre – sortedList is sorted
    - Post – sortedList is sorted, has same elements as before plus the new one
  + Remove:
    - Pre – sortedList is sorted & indicated element is present
    - Post – sortedList is sorted, has same elements as before minus the indicated one
      * But what if element appears more than once??
* **Priority queue** – like a queue, but each element has a value & an integer priority
  + Enter – “same as before” (can’t tell difference until leave is called)
  + Leave – remove the oldest element with the lowest/highest priority
  + New node type:

Struct Node {

string val;

int priority;

Node\* next;

};

* List of list implemenetation
  + Insert: find the right queue for priority (O(k) time for k priorities)
    - Insert into queue (O(1) time)
    - Total time for insertion – O(k) time
  + Usually implemented as a heap
    - Lookup, insert, remove are O(log n); peek is O(1)