School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

Linked List

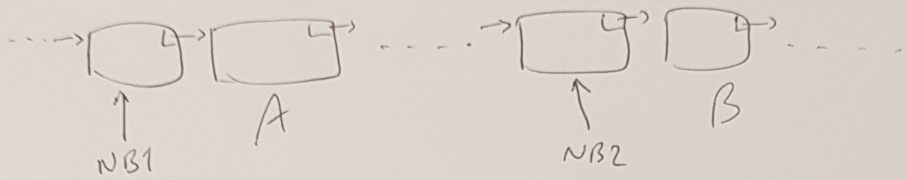
Review

- Linked lists
  - Used for implementing stacks and queues
- Using singly linked lists we saw
  - Simple operations on a list
  - Search

- Today’s topic:
  - Swap two nodes
  - Sort (not in detail)
  - Doubly linked lists
  - Priority Queue

Swap 2 nodes of a Linked Lists

We want to swap A & B. We Assume A is not the first node.




```
void swap(Node* NB1, Node* NB2)
{
    Node* temp1 = NB1->next->next;
    NB1->next->next = NB2->next->next;
    NB2->next->next = temp1;

    Node* temp2 = NB1->next;
    NB1->next = NB2->next;
    NB2->next = temp2;

}
```

Sorting Linked Lists

- If you are ready! let's see how a linked list can be sorted!

- Insertion sort
  - Same as array, but we need to traverse from the beginning to find the appropriate position of each new item
- Selection sort
  - Pretty much the same as on an array
- Think about
  - Merge sort (this is good for a linked list)
    - O(n) for splitting and merging, so it can we written as T(n/2) +O(n)
  - Quick sort (needs traversing from end as well)
    - Complexity?

Common mistakes

- Referring through a linked list, without checking for non-existent entries.
- Unless you know what you're doing, this is very likely to fail, for a Node *temp:
  int x = temp->link->link->data
  What if temp->link is NULL?

- Don't build complex testing conditions where a simple approach will work better.

- Common pitfall: trying to copy the list by setting newHead = head or newList=myList
  - This creates two pointers to the same head, not a copy of the list.
  - The second one works if you have provided a **copy constructor** for the linked list class and make a copy of each node it that

Add a node to a linked list

- Insert at the beginning
  - O(1)
- Insert at end
  - O(n)
  - Keep a pointer to tail?
    - O(1)
- Insert before tail
  - O(n)
  - Doubly linked list
    - O(1)
- Insert in middle for singly or doubly linked lists
  - O(n)

Doubly linked lists

- If we add a prev pointer to our node, we can walk in either direction. Now our node would be:

```
struct Node {
    type data;
    Node* next;
    Node* prev;
};
```

- Memory usage
- Lets see how to insert a new node in the middle

When to use doubly linked list

- You can
  - Traverse in both directions, think of
    - deleting a node
    - swapping two nodes
    - implementing a queue with a doubly linked list
    - Quick sort
- But
  - With more trouble and time (constant factor) for updating the links
  - And more memory usage

- Use doubly linked lists if you feel the need to traverse the list in opposite direction pretty often.
- Or when most operations happen often around the end of the list

Queue Example: Priority Queues

- A lot of useful queues have a notion of priority associated with them.
- Some people/processes will take precedence over others for a variety of reasons.
- This happens in networking, operating systems, printing and real life (if you're famous in America and trying to get into a restaurant).

Queue Example: Priority Queues

- There are at least two ways to approach a priority queue. The fundamental functions (enqueue and dequeue functions) that we need to support are
  - remove_highest and simple add
    - Complexity
      - O(n), O(1)
  - add_with_priority and simple remove
    - Complexity (linked list)
      - O(n) and O(1)
    - Complexity (array)
      - Add O(log n)? Or O(n)?, remove O(1)
  - Given a queue (add, remove and isEmpty only!) how do you do it?
- What happens if everything has the same priority?
- Other solution: keep several queues (linked list- array)

Implications of structure

- What if you want to insert an item in the middle of stacks and queues?

- We can only 'see' the top element (for a stack) or the front element (for a queue)
- These abstract data types provide very different ways of interacting with data than that of a simple array or Linked List.

No free walks

- Stacks have push, pop and empty as their basic operations.
- Queues have add, remove and empty (or enqueue, dequeue sometimes instead of add and remove)

- If you need an element in the middle, you need to pop or dequeue all elements that come before it
- Same situation if you need to insert an element in the middle (both are easier for a queue – no need for an auxiliary list)

Stacks in C++

- Stacks have a defined top of stack
  – Insertion and deletion are efficient as they occur at a single point - only one pointer has to be maintained.
- Top returns the value and pop removes it

```
#include <stack>            // std::stack
int main ()
{
    std::stack<int> mystack;
    mystack.push(5);
    if (!mystack.empty())
    {
        std::cout << mystack.top(); //does not remove
        mystack.pop(); //does not return
    }
}
```

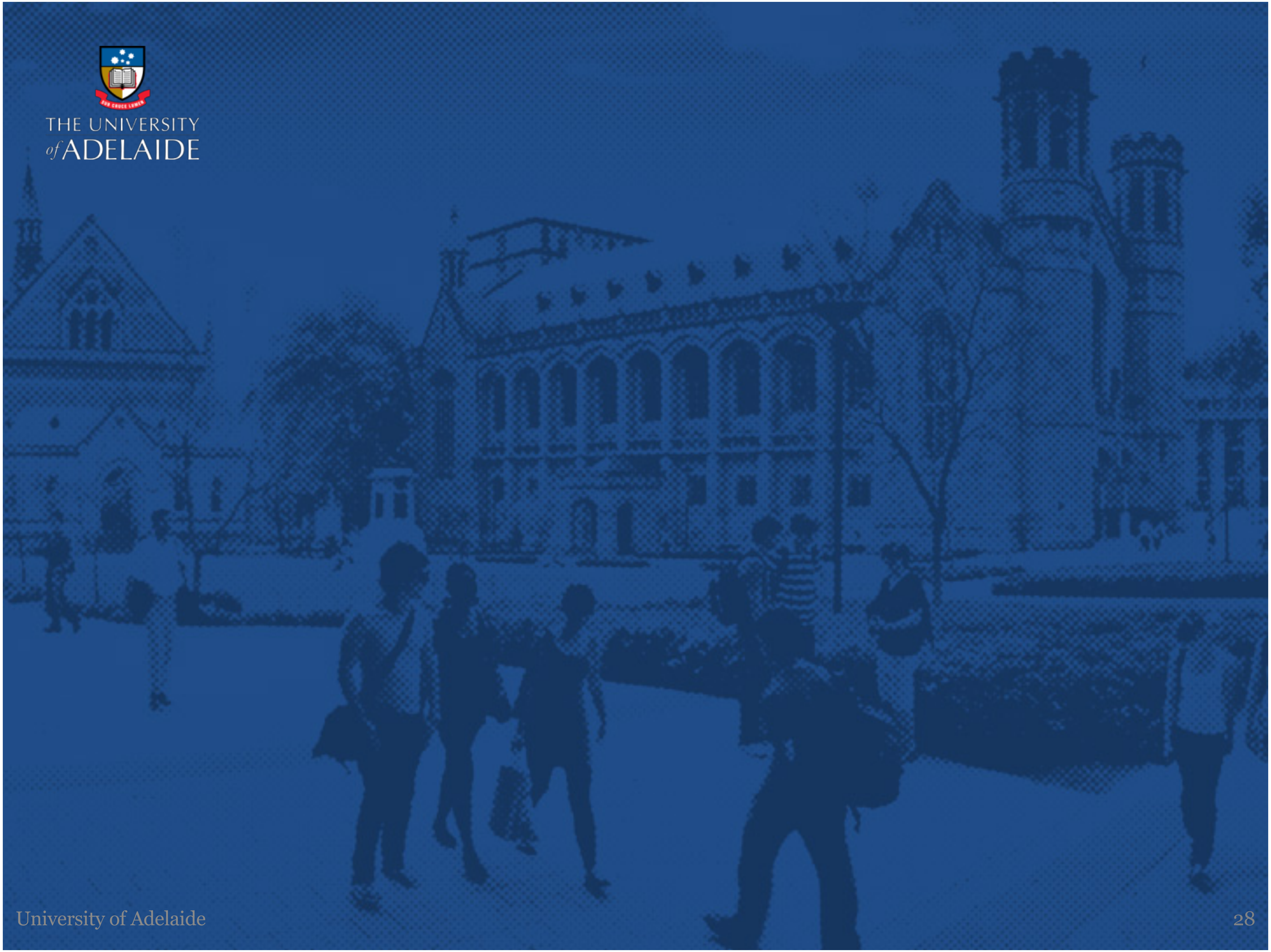
Queues in C++

- Queues have a defined front and back
  - Insertion and deletion are efficient as they occur at well-defined points - two pointers have to be maintained for front and back.
- Front returns the value and pop removes it
- Queues are easy to understand and explain.

```
#include <queue>          // std::queue
int main ()
{
    std::queue<int> myqueue;
    myqueue.push (5);
    if (!myqueue.empty());
    {
        std::cout << ' ' << myqueue.front();
        myqueue.pop();
    }
}
```

Efficient Use

- Searching a stack or standard queue for a value requires 2 * O(n) operations to take everything out, look at it, and put it back again.
- We should design with these ADTs if the add and remove operations are efficient for our purposes.
- Not everything should be put into a stack or a queue! Depending on your requirements, you may need to use other structures.


THE UNIVERSITY
of ADELAIDE