School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

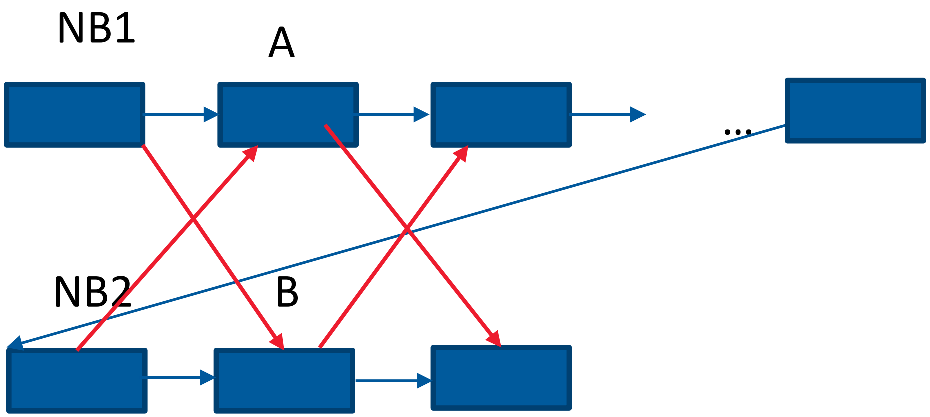
More Functions on Linked Lists, Stacks and Queues

Review

- Linked lists
  - Used for implementing stacks and queues
- Using singly linked lists we saw
  - InsertAtFront
  - Search
  - InsertAfter

- This lecture:
  - Swap 2 nodes
  - Delete

Swap 2 nodes of a linked lists



Swap A and B.
Assume that A and B are not the first node. Let NB1 and NB2 be the predecessors of A and B, respectively.

```
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;
}
```

Deleting a node

- When deleting a node we have to make sure that we have copied out its link pointer before we delete it, or everything from that point on in the chain is lost forever
- Here’s some sample code, let’s say we want to delete node discard, where the list is:
  - head->before->discard->…

```
before->link = discard->link;
delete discard;   // Clean up
```

Pointers as iterators

- An iterator is a construct that allows you to cycle through the data items stored in a data structure so that you can perform whatever action you want on each data item.

```
Node_Type *iter;
for(iter = head; iter != NULL; iter = iter->link)
  // do whatever you want on each item
```

Common mistakes

- Referring through a linked list without checking for non-existent entries:
  ex. for a Node *temp:

  `int x = temp->link->link->data`
- Building complex testing conditions where a simple approach works better.
- 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

Advantages of the linked list

- Linked lists are easy to understand and explain.
- Insertion and deletion are efficient.
- We only allocate space as we need it.

- What about dynamic array?
  - Dynamic arrays can fill up.
  - A good implementation will expand as required, but to be efficient, the expansion occurs in fixed size blocks.

Efficient linked lists

- To be efficient, we must ensure that:
  - We only create as many elements as we need
  - We insert and delete correctly
  - We delete the elements removed from the list.
  - We use an appropriate pointer strategy
  - Keep track of the head

Some structures are more complex

- A linked list may consist of nodes that are, in turn, the heads of other linked lists.

```
struct Node {
int data;
Node *link;
Node *otherLink;
}
```

- This would form a list of lists.

Some last words

- Linked lists are linear structures.
- You can draw them in a straight line, one after the other.
- There are no gaps.
- When doubt, draw the structure to see how it should work.

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;
};
```

When to use doubly linked list

- Need to traverse the list in opposite direction.

- With doubly linked list, you can traverse the list in both directions, but with
  - more trouble and time for updating the links
  - more memory usage

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 in middle
  - O(n)
- Insert before a given pointer
  - Singly O(n)
  - Doubly O(1)

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

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

Implications of structure

- 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 isEmpty as their basic operations.
- Queues have enqueue, dequeue and isEmpty
- 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

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.

Queue example: priority queue

- 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 queue

- There are at least two ways to approach a priority queue. The fundamental functions that we need to support are enqueue and dequeue.
- Options
  - remove highest and simple add
    - Complexity
      - O(n), O(1)
  - add with priority and simple remove
    - Complexity
      - O(n) and O(1)
- Other solution: keep several queues

THE UNIVERSITY of ADELAIDE