CS122 Algorithms and Data Structures

MW 11:00 am - 12:15 pm, MSEC 101
Instructor: Xiao Qin
Lecture 5: Doubly Linked Lists and Linked Stacks

Lists
- A list is a varying-length, linear collection of homogeneous elements.
- Linear means each list element (except the first) has a unique predecessor, and each element (except the last) has a unique successor.
- In a linear structure, components can only be accessed sequentially one after the other.

An Insert Algorithm for Sorted Lists
- Sorted list: list elements sorted in ascending order.
- How would the algorithm to insert an item into a sorted linked list?

class SortedList

Transformers
- InsertTop
- Insert
- DeleteTop
- Delete

Observers
- Print
- IsEmpty

ADT SortedList Operations

struct NodeType

// SPECIFICATION FILE DYNAMIC-LINKED SORTED LIST (slist.h)
typedef int ItemType; // Type of each component
// Is simple type or string type
struct NodeType
{
  ItemType item; // Pointer to person's name
  NodeType* next; // Link to next node in list
};
typedef NodeType* NodePtr;
Insert algorithm for SortedList

- find proper position for the new element in the sorted list using two pointers prevPtr and currPtr, where prevPtr trails behind currPtr
- obtain a node for insertion and place item in it
- insert the node by adjusting pointers

Finding Proper Position for 12

```
while (currPtr != NULL && item > currPtr->info)
{
    prevPtr = currPtr;
    currPtr = currPtr->next;
}
```

Inserting '12' into Proper Position

```
location->next = currPtr;
if (prevPtr == NULL)
    head = location;
else
    prevPtr->next = location;
```
Doubly Linked Lists

- An extension of a Singly Linked List
- Each node has two pointers
  - One pointing to the successor
  - One pointing to the predecessor
- They are used because they ease certain operations like the deleteElement
- They are interesting for traversal as you can move in either directions

Doubly Linked Lists (Cont.)

- Most of our structures and algorithms will be implemented with the help of Doubly Linked Lists
- Although some operations are made easier to understand they also become a bit slower due to the overhead of extra pointers
- In addition to the head a pointer called tail is also maintained
Inserting in the Front

```
NodeType *temp = new NodeType;
temp->data = 8;
temp->next = head;
head->prev = temp;
head = temp;
temp = NULL;
```

Inserting in the Front (cont.)

```
NodeType *temp = new NodeType;
temp->data = 8;
temp->next = head;
head->prev = temp;
head = temp;
temp = NULL;
```

Inserting in the Front (cont.)

```
NodeType *temp = new NodeType;
temp->data = 8;
temp->next = head;
head->prev = temp;
head = temp;
temp = NULL;
```

Inserting in the Front (cont.)

```
NodeType *temp = new NodeType;
temp->data = 8;
temp->next = head;
head->prev = temp;
head = temp;
temp = NULL;
```

Deleting element ‘12’

```
NodeType *current = head;
while (current->data != 12 and current->next != NULL)
    current = current->next;
}
current->prev->next = current->next;
current->next->prev = current->prev;
delete current;
```
Deleting element ‘12’ (cont.)

```c
NodeType *current = head;
while (current->data != 12 and current->next != NULL)
    current = current->next;
if (current->data == 12)
    {
        current->prev->next = current->next;
        current->next->prev = current->prev;
        delete current;
    }
```

Circular Lists
- Nodes form a ring
- Each node has a successor

Inserting a Node at the Tail of a List
```c
int item = 6;
Node *location;
location = new NodeType;
location->info = item;
tail->next = location;
tail = tail->next;
```
Inserting a Node at the Tail of a List
Circular Lists (cont.)

```
int item = 6;
Node *location;
location = new NodeType;
location->info = item;
tail->next = location;
tail = location;
```

Inserting a Node at the Tail of a List
Circular Lists: A Special Case

```
int item = 6;
Node *location;
location = new NodeType;
if (isEmpty()) {
tail = location;
tail->next = tail;
}
```

Debugging Linked Lists

**Two Suggestions Only**

- Draw your linked list.
- Consider special cases.
Skip Lists
- Linked lists require sequential scanning for a search operation
- Skip lists allow for skipping certain nodes
- A skip list is an ordered linked list

A Find Algorithm for Skip Lists
\[
\text{Find}(element \, el) \\
p = \text{the non-null list on the highest level} \, i; \\
\text{while} \, el \, \text{not found and} \, i > 0 \\
\quad \text{if} \, p->\text{key} > el \quad \text{//eg.} \, el = 8 \\
\quad \quad p = \text{a sublist that begins in} \, p \, \text{on level} \, i--; \\
\quad \quad \text{if} \, p->\text{key} = el \\
\quad \quad \quad \text{if} \, p \, \text{is the last node on level} \, i; \quad \text{//eg.} \, el = 16 \\
\quad \quad \quad \quad p = \text{a nonnull sublist begins in} \, p \, \text{on the highest level} < i; \\
\quad \quad \quad \quad i = \text{the number of the new level}; \, i = i-1 \\
\quad \quad \quad \text{else} \, p = p->\text{next}; \\
\quad \quad \text{return(TRUE);} \\
\quad \text{else return(FALSE);} \\
\]

Stacks using Linked Lists
- Implementation of stacks using linked lists are very simple
- The difference between a normal linked list and a stack using a linked list is that some of the linked list operations are not available for stacks
- Being a stack we have only one insert operation called \text{push()}
  - In many ways \text{push} is the same as \text{insert} in the front
- We have also one delete operation called \text{pop()}
  - This operation is the same as the operation delete from the front
- The other important operations in a stack, called \text{top()} and \text{isEmpty()}, don't modify the structure
- To implement stacks we will use a singly linked list

A series of operations executed on a stack
- \text{push 3, push 7, pop, push 1, push 4, pop, pop}
push(45)

```c
NodeType *temp = new NodeType;
temp->data = n;
temp->next = top;
top = temp;
temp = NULL;
```

push(45)

```c
NodeType *temp = new NodeType;
temp->data = n;
temp->next = top;
top = temp;
temp = NULL;
```

push(45)

```c
NodeType *temp = new NodeType;
temp->data = n;
temp->next = top;
top = temp;
temp = NULL;
```

pop()

```c
NodeType *temp = top;
top = top->next;
int returnValue = temp->data;
delete temp;
return returnValue;
```

pop()

```c
NodeType *temp = top;
top = top->next;
int returnValue = temp->data;
delete temp;
return returnValue;
```
pop()

NodeType *temp = top;
top = top->next;
int returnValue = temp->data;
delete temp;
return returnValue;

Value stored in the top has to be returned before node can be deleted.

pop()

NodeType *temp = top;
top = top->next;
int returnValue = temp->data;
delete temp;
return returnValue;

top()

return top->data;