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

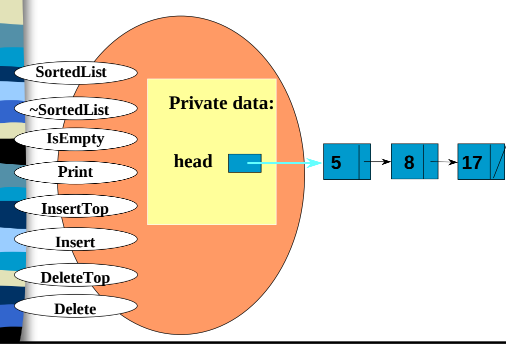
2

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?

3

`class SortedList`



4

ADT SortedList Operations

Transformers
- InsertTop
- Insert
- DeleteTop
- Delete

change state

Observers
- Print
- IsEmpty

observe state

5

`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;
```

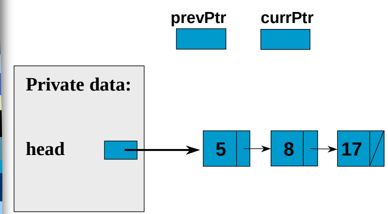
6

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**

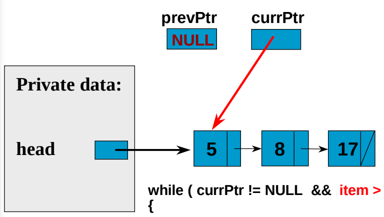
7

Inserting 12 into a Sorted List



8

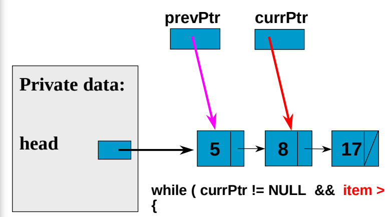
Finding Proper Position for 12




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

9

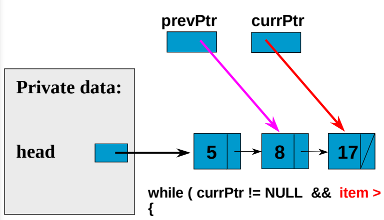
Finding Proper Position for 12




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

10

Finding Proper Position for 12

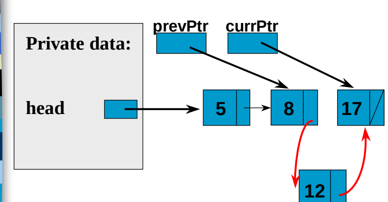



```
while ( currPtr != NULL  &&  item > currPtr->info  )
{
        prevPtr = currPtr ;
        currPtr = currPtr->next ;
}
```

11

Inserting ’12’ into Proper Position

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



12

```
// IMPLEMENTATION DYNAMIC-LINKED SORTED LIST
// (slist.cpp)
SortedList ::SortedList ( )          // Constructor
// Post:    head == NULL
{
    head = NULL ;
}

SortedList2 :: ~SortedList2 ( )     // Destructor
// Post:  All linked nodes deallocated
{
    ItemType  temp ;
                                     // keep deleting top node
    while  ( !IsEmpty )
          DeleteTop ( temp );
}
```

13

```
void  SortedList :: Insert( ItemType  item )
//  Pre:    item is assigned &&  list components in ascending order
// Post:  new node containing item is in its proper place
//          && list components in ascending order
{    NodePtr     currPtr, prevPtr, location ;
     location = new  NodeType ;
     newNodePtr->info  =  item ;
     prevPtr = NULL ;
     currPtr = head ;
     while ( currPtr != NULL  &&  item > currPtr->info  )
     {      prevPtr = currPtr ;          // advance both pointers
            currPtr = currPtr->next ;
     }
     location->next = currPtr ;      // insert new node here
     if  ( prevPtr == NULL )
            head = location ;
     else
            prevPtr->next = location ;
}
```

14

```
void SortedList :: DeleteTop ( ItemType&  item )
//  Pre:    list is not empty && list elements in ascending order
// Post:   item == element of first list node @ entry
//         && node containing item is no longer in linked list
//         && list elements in ascending order
{
    NodePtr  tempPtr = head ;

    item = head->info  ;           // obtain item
    head = head->next ;           // advance head
    delete  tempPtr ;
}
```

15

```
void  SortedList :: Delete ( /* in */  ItemType  item )
//  Pre:      list is not empty && list elements in ascending order
//        &&  item == component member of some list node
// Post:     item == element of first list node @ entry
//        &&  node containing first occurrence of item is no longer
//            in linked list   &&  list elements in ascending order
{    NodePtr   delPtr ;
     NodePtr   currPtr ;                 //  Is item in first node?
     if ( item == head->info )
     {      delPtr = head ;               // If so, delete first node
            head = head->next ;
     }
     else {                               // search for item in rest of list
            currPtr = head ;
            while ( currPtr->next->info  !=  item )
                    currPtr = currPtr->next ;
            delPtr = currPtr->next ;
            currPtr->next = currPtr->next->next ;
     }
     delete  delPtr ;
}
```

16

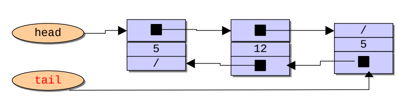
Doubly Linked Lists

- An extension of a Singly Linked List
- Each node has two pointer
  - 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

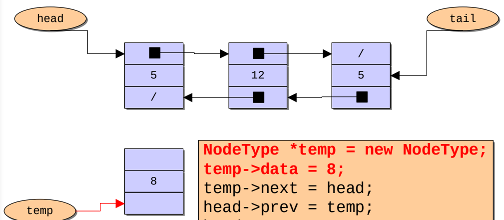
```
struct NodeType {
    <someType> data;
    NodeType* prev;
    NodeType* next;
}
```

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

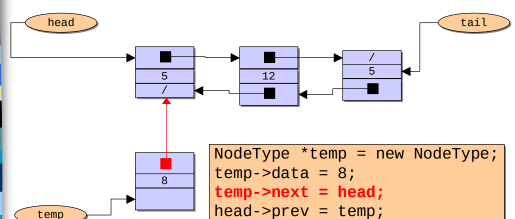


Inserting in the Front



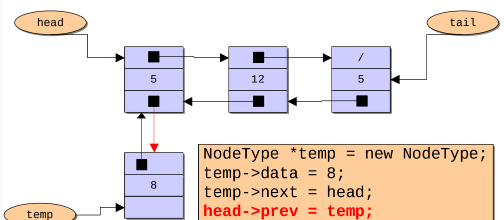

```
NodeType *temp = new NodeType;
temp->data = 8;
temp->next = head;
head->prev = temp;
head = temp;
temp = NULL;
```

Inserting in the Front (cont.)



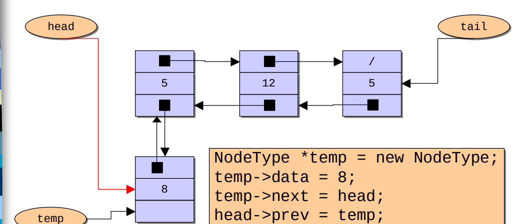

```
NodeType *temp = new NodeType;
temp->data = 8;
temp->next = head;
head->prev = temp;
head = temp;
temp = NULL;
```

Inserting in the Front (cont.)



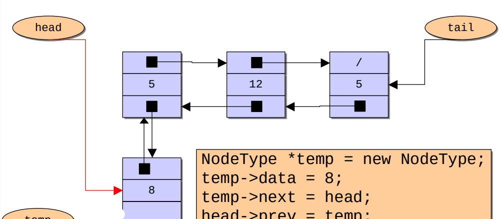

```
NodeType *temp = new NodeType;
temp->data = 8;
temp->next = head;
head->prev = temp;
head = temp;
temp = NULL;
```

Inserting in the Front (cont.)



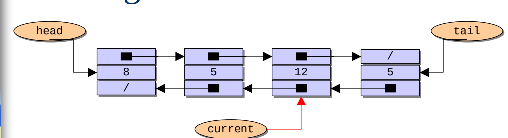

```
NodeType *temp = new NodeType;
temp->data = 8;
temp->next = head;
head->prev = temp;
head = temp;
temp = NULL;
```

Inserting in the Front (cont.)



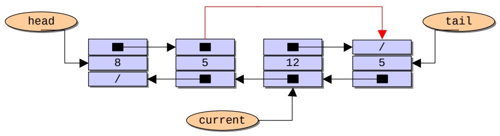

```
NodeType *temp = new NodeType;
temp->data = 8;
temp->prev = head;
temp->next = head;
head->prev = temp;
head = temp;
temp = NULL;
```

Deleting element ’12’



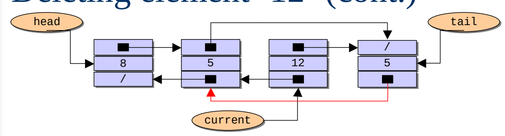

```
NodeType *current = head;
while (current->data != 12 and current->next != NULL)
    current = current->next;
}
current->prev->next = current->next;
current->next->prev = current->prev;
delete current;
```

Deleting element '12' (cont.)



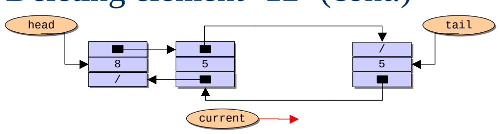

```
NodeType *current = head;
while (current->data != 12 and current->next != NULL)
    current = current->next;
}
current->prev->next = current->next;
current->next->prev = current->prev;
delete current;
```

Deleting element '12' (cont.)




```
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;
```

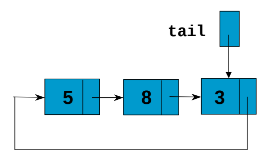
Deleting element '12' (cont.)




```
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;
```

Circular Lists

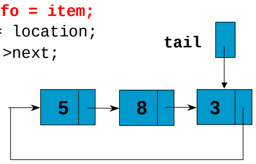
- Nodes form a ring
- Each node has a successor

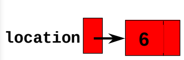


28

Inserting a Node at the Tail of a List Circular Lists

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

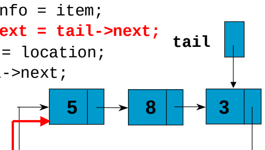


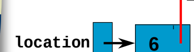


29

Inserting a Node at the Tail of a List Circular Lists (cont.)

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

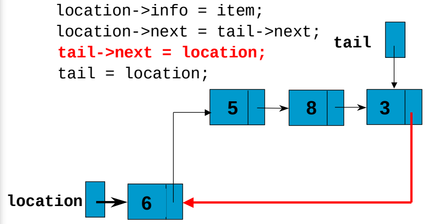




30

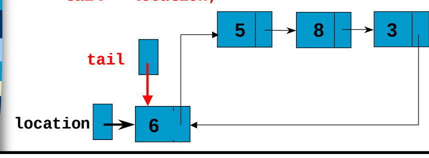
Inserting a Node at the Tail of a List Circular Lists (cont.)

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



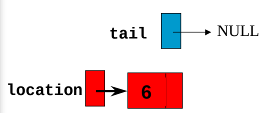
Inserting a Node at the Tail of a List Circular Lists (cont.)

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



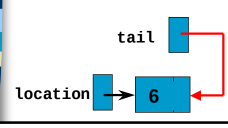
Inserting a Node at the Tail of a List Circular Lists: A Special Case

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



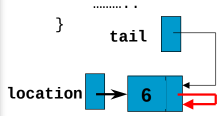
Inserting a Node at the Tail of a List Circular Lists: A Special Case (cont.)

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



Inserting a Node at the Tail of a List Circular Lists: A Special Case (cont.)

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



Debugging Linked Lists

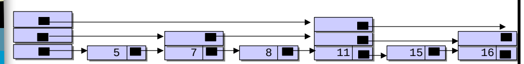
Two Suggestions Only

- Draw your linked list.
- Consider special cases.

Skip Lists

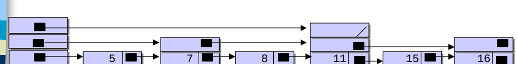
- Linked lists require sequential scanning for a search operation
- Skip lists allow for skipping certain nodes
- A skip list is an ordered linked list

A Skip List with Evenly spaced Nodes of different levels



A Find Algorithm for Skip Lists

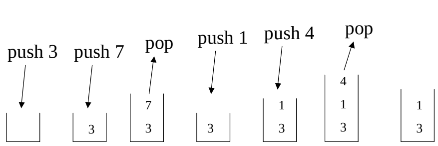
```
Find(element el)
   p = the non-null list on the highest level i;
   while el not found and level i >= 0
      if p->key > el      //eg. el = 8
         p = a sublist that begins in the predecessor of p on level i--;
      else if p->key < el
         if p is the last node on level i;    //eg. el = 16
            p = a nonnull sublist begins in p on the highest level < i;
            i = the number of the new level; //i = i-1
         else p = p->next;
      else return(TRUE);
   return(FALSE);
```



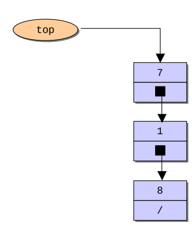
Stacks using Linked Lists

- 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 `push()`.
  - In many ways push is the same as insert in the front
- We have also one delete operation called `pop()`
  - This operation is the same as the operation delete from the front
- The other important operations in a stack, called `top()` and `isEmpty()`, don't modify the structure
- To implement stacks we will use a singly linked list

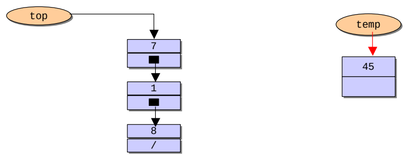
A series of operations executed on a stack



Pictorial view of a stack

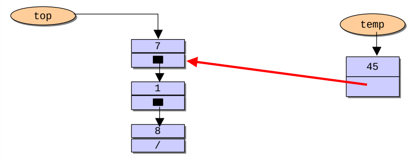


push(45)



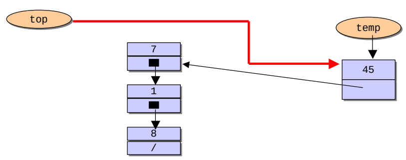

```
NodeType *temp = new NodeType;
temp->data = n;
temp->next = top;
top = temp;
temp = NULL;
```

push(45)



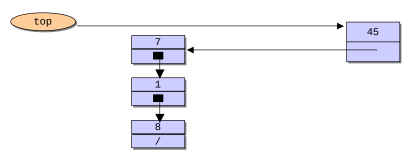

```
NodeType *temp = new NodeType;
temp->data = n;
temp->next = top;
top = temp;
temp = NULL;
```

push(45)



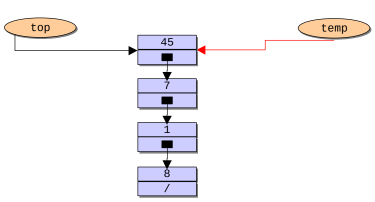

```
NodeType *temp = new NodeType;
temp->data = n;
temp->next = top;
top = temp;
temp = NULL;
```

push(45)



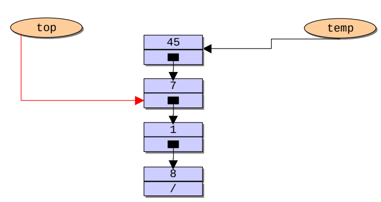

```
NodeType *temp = new NodeType;
temp->data = n;
temp->next = top;
top = temp;
temp = NULL;
```

pop()



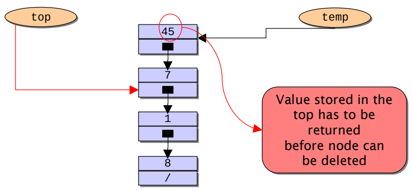

```
NodeType *temp = top;
top = top->next;
int returnValue = temp->data;
delete temp;
return returnValue;
```

pop()



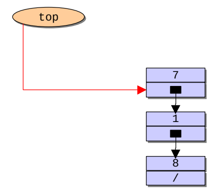

```
NodeType *temp = top;
top = top->next;
int returnValue = temp->data;
delete temp;
return returnValue;
```

pop()



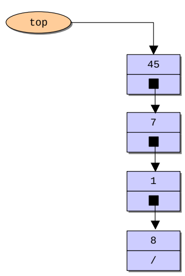

```
NodeType *temp = top;
top = top->next;
int returnValue = temp->data;
delete temp;
return returnValue;
```

pop()




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

top()




```
return top->data;
```