ECE 2620: C++, Data Structures & Algorithms
Linked Lists Using Pointers

Dr. Sarvesh Kulkarni

Department of Electrical & Computer Engineering
Villanova University, Villanova, PA 19085
Email: sarvesh.kulkarni@villanova.edu

Table of Contents

1. Singly Linked Lists (SLLs)
   - SLL Node Class & Its Use
   - SL List Class & Its Member Functions
2. Doubly Linked Lists (DLLs)
   - DLL Node Class
   - DL List Class & Its Member Functions
3. Circular Lists: CSLLs & CDLLs

Linked Lists

Disadvantages of Automatic Arrays (non-dynamically allocated):

1. Array size must be known at compile time
2. Insertions, deletions (with compacting), reordering of array elements is computationally expensive

Disadvantage 1 is overcome by using dynamically allocated arrays

But that still does not address disadvantage 2!

However, both disadvantages are overcome if we implement **linked lists** ..

- As either **singly**, or **doubly** linked lists
- With **dynamic** data structures

Singly & Doubly Linked Lists

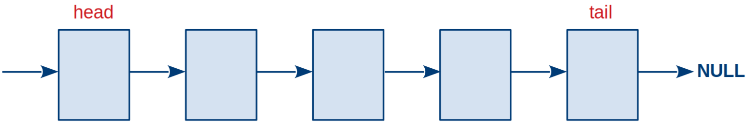


Figure: Singly Linked List (SLL): Only forward traversal

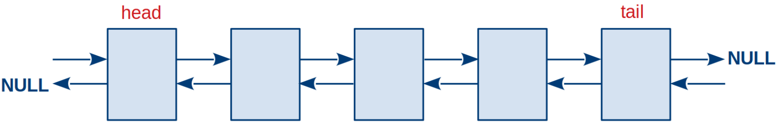


Figure: Doubly Linked List (DLL): Forward & backward traversal

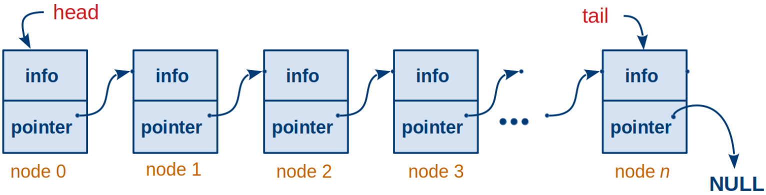
Singly Linked Lists

Let us construct a list of integers (to replace an array of integers), implemented as an SLL

Suppose our array was previously defined like so:

```
int info[MAXSIZE];
```

Then, the equivalent SLL would look like this:



List nodes do not have indices since this is not an array

SLL Nodes are implemented in the class shown on the next slide

A Class for SLL Nodes (in header file sllnode.h)

```
#ifndef _SLL_NODE
#define _SLL_NODE
class intSLLNode {  // Objects of this class are nodes in our SLL
     friend class intSLList; //so that we may access node members
                             // in intSLList class that follows
     public:
             intSLLNode(int el=0, intSLLNode *ptr=nullptr) {
                 info = el;
                 next = ptr;
             }

     protected:
             int         info;   // int type data member
             intSLLNode *next;  // pointer type data member
};
#endif
```

Data members may be declared as public, but it is better to declare them as protected. If they are protected, we must make intSLList class our friend; else our intSLList class cannot access intSLLNode's data members

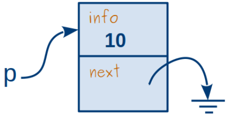
Stringing SLL Nodes Into a List (1)

We can now get the list started:

1.
```
// Declare pointer p capable of pointing to intSLLNode
intSLLNode *p;
```

2.
```
// Dynamically create new node and make p point to it
// And, initialize info to 10 using class constructor
p = new intSLLNode(10);
```

The above two steps result in a single-node list:



(1)

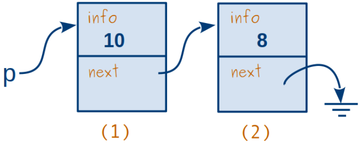
To initialize 'info' and 'next' fields, we could have written a member function, instead of using the constructor (implicitly)

Stringing SLL Nodes Into a List (2)

Continuing with the example from the earlier slide ..

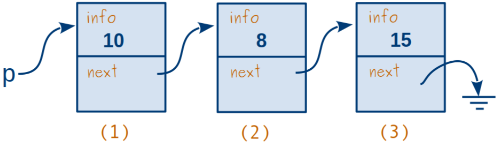
3.
```
// Let us add another node with value 8 in the 'info' field:
p->next = new intSLLNode(8);
```

We get:



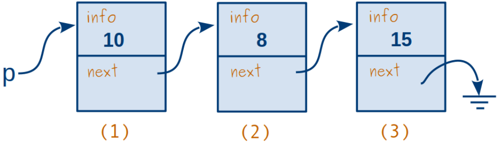
4.
```
// Adding a third node with value 15 in the 'info' field:
p->next->next = new intSLLNode(15);
```

We get:



Stringing SLL Nodes Into a List (3)

Now we have three nodes in our list ..



```
p                       // points to (1)
p->next                 // points to (2), same as (*p).next
(p->next)->next         // points to (3), = (*((*p).next)).next
((p->next)->next)->next // points to NULL

p->info                 // has value of 10, same as (*p).info
(p->next)->info         // has value of 8, = (*((*p).next)).info
((p->next)->next)->info // has value of 15

(((p->next)->next)->next)->info   // is an invalid reference
```

A Quick Reminder

The following two statements:

```
intSLLNode *ptr;    // declare pointer ptr of type intSLLNode
ptr = nullptr;      // point ptr to NULL address
```

may be replaced by the single statement:

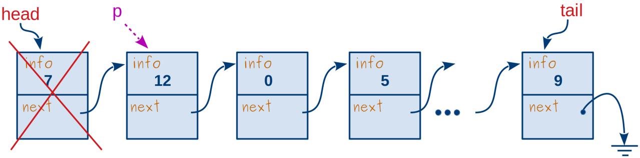
```
intSLLNode *ptr = nullptr; // declare pointer ptr of type
                           // intSLLNode AND point it to NULL
                           // address
```

To simplify matters and to promote data abstraction, we now define the class **intSLList** on the next slide

Class To Manage List of SLL Nodes (in header file sllist.h) (1)

```
#ifndef _SLL
#define _SLL
// An object of this class is a list of nodes of type intSLLNode
// defined in slide 6
class intSLList  {
     public:
           intSLList() {head= tail= nullptr;}
           ~intSLList();
           bool isEmpty() {return (head == nullptr);}
           void addToHead(int);
           void addToTail(int);
           int deleteFromHead();  //delete head & return its info
           int deleteFromTail();  //delete tail & return its info
           void deleteNode(int);
           bool isInList(int) const;
     private:
           intSLLNode  *head, *tail;
};
#endif
```

The **intSLList** Class's **Destructor** (2)



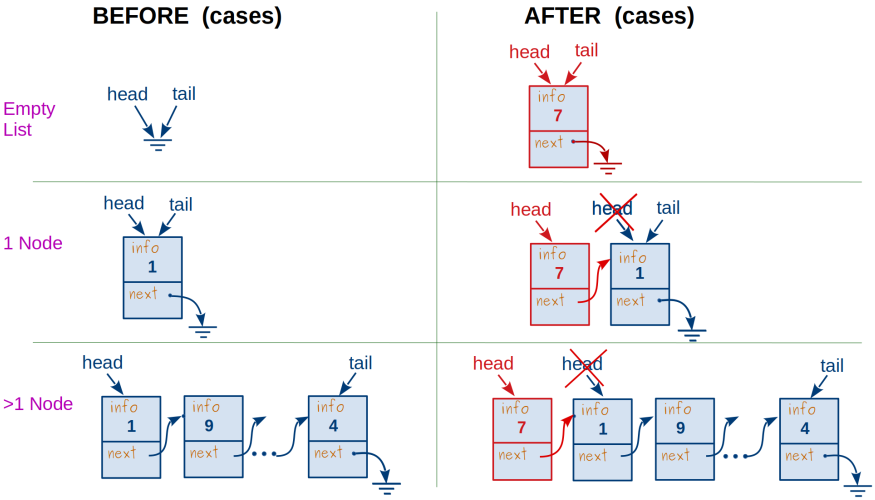
Strategy:

As long as the list is not empty, repeat the following:

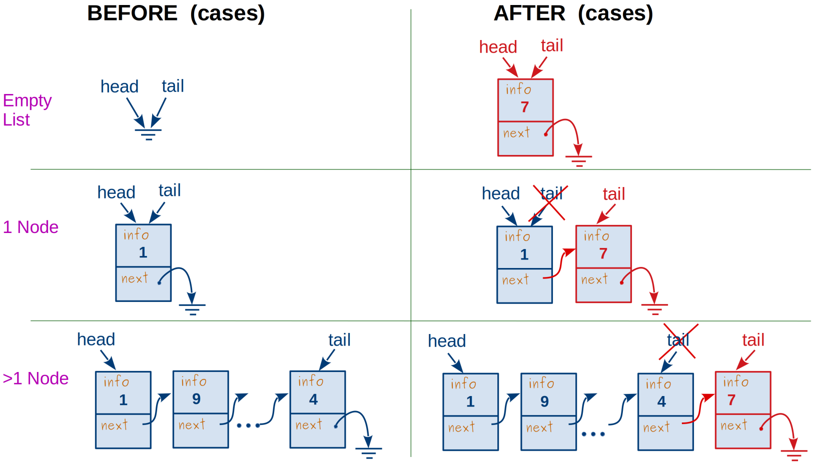
1. Place pointer p one step to the right of the head node
2. Delete head node
3. If list is not empty, repeat steps 1 – 2, otherwise stop

```
intSLList::~intSLList() {
      for (intSLLNode *p; !isEmpty(); head = p;) {
          p = head->next;
          delete head;
      } // for
      ~~tail = nullptr;~~  // No need to set tail ptr to NULL; WHY?
}
```

Strategy Sketch for Member Function **addToHead** (3)



Strategy Sketch for Member Function **addToTail** (4)



Member Functions **addToHead** & **addToTail** (5)

All member functions go in the file sllist.cc

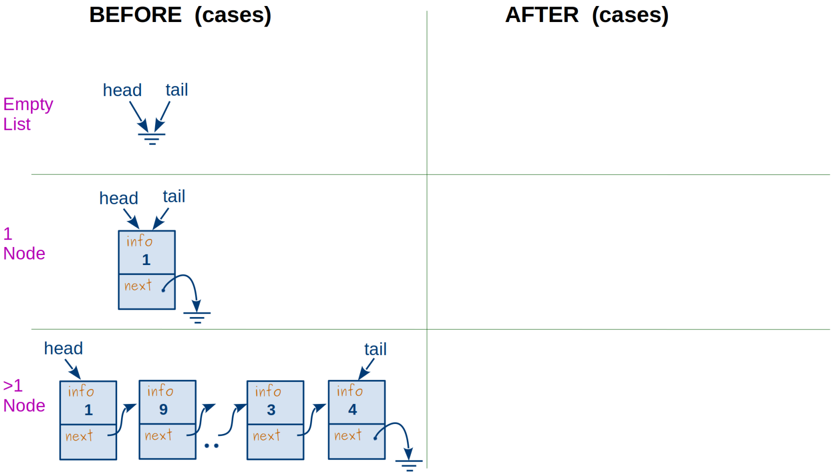
The code: the **addToHead** member function

```
void intSLList::addToHead(int el) {
    head = new intSLLNode(el,head);   // Understand this
    if (tail == nullptr)  tail = head;
}
```

The code: the **addToTail** member function

```
void intSLList::addToTail(int el) {
    if (tail != nullptr) {   // if list is not empty
        tail->next = new intSLLNode(el);
        tail = tail->next;
    }
    else head = tail = new intSLLNode(el);
}
```

Strategy: The “Before” & “After” Sketches (6)



BEFORE (cases)

AFTER (cases)

Empty List

1 Node

>1 Node

Figure: Use this template to sketch your strategy

Member Function **deleteFromHead** (7)

Sketch your strategy first; use template on previous slide

```
int intSLList::deleteFromHead() {
    if(!isEmpty()) {
       int el = head->info;
       intSLLNode *tmp = head;
       if (head == tail)  // i.e. if only 1 node in list
           head = tail = nullptr;
       else
           head = head->next;
       delete tmp;
       return (el);
    }
    else throw(EMPTY_LIST);
    // where, EMPTY_LIST is declared (somewhere globally) as ..
    // const int EMPTY_LIST = 1;
}
```

throw *throws* an **exception** called "EMPTY_LIST". A try-catch clause around the fn call *must "catch"* this exception; see next slide

try-catch Clause in Fn Call to **deleteFromHead** (8)

```
void foo() {  // or, int main()
      : :
    int val;
    try {
         val = list.deleteFromHead();  // call to member fn
        } catch (int error_code)
             {
              cerr << " Error: " << error_code << endl;
              switch (error_code)
                { // DO SOMETHING IN RESPONSE TO THE ERROR HERE
                    : :
                }
             } // catch ends
      : :
}
```

NOTE: cerr is **unbuffered** output, unlike cout

If the try-catch exception handler is missing, a thrown exception will cause a program crash. The programmer must handle the exception gracefully once it is caught!

Member Function **deleteFromTail** (9)

Sketch your strategy first; use template on *slide 16*

```
int intSLList::deleteFromTail() {
    if(!isEmpty()) {
        int el = tail->info;
        if (head == tail) { // i.e. if only 1 node in list
            delete tail;
            head = tail = nullptr;
        }
        else {  // i.e if there is more than 1 node in the list
            intSLLNode *tmp;
            // find predecessor of tail before deleting tail
            for (tmp=head; tmp->next!=tail; tmp=tmp->next);
            delete tail;
            tail = tmp;  // the predecessor of tail becomes tail
            tail->next = nullptr;
       }
       return (el);
    }
    else throw(EMPTY_LIST);
}
```

Member Function **isInList** (10)

```
bool intSLList::isInList(int el) const {
    intSLLNode *tmp;
    for (tmp = head; tmp != nullptr && tmp->info != el;
         tmp = tmp->next);
    return (tmp != nullptr);
}
```

NOTE:

The const keyword (in the 1st line, after the input parameter list) disallows this member fn from altering any data members (i.e. head & tail pointers) in class intSLList

In short, the data members of this class are treated as constants in this member fn.

Member Function **isInList** (11)

Observations on this slide refer to the example from the previous slide

The statement:

```
return (tmp != nullptr);
```

may also be written as:

```
if (tmp != nullptr)  return  true;
else                 return false;
```

We cannot replace the code above (to determine whether the element has been found), with the following code:

```
if (tmp->info == el)  return true;
else                  return false;
```

because it will cause a seg-fault if the list is empty, or if the element is not found (since, in those cases, tmp points to a NULL address after the for loop terminates)

Member Function **deleteNode** (12)

```
void intSLList::deleteNode(int el) {
    if (head != nullptr) // if nonempty list
        if (head == tail && el == head->info) { // 1 node in list
            delete head;
            head = tail = nullptr;
        }
        else {
           if (el == head->info) {   // if > 1 node is in list
             intSLLNode *tmp = head; // and if el is in head node
             head = head->next;      // then advance  head ptr
             delete tmp;             // and delete old head node
           }
           else { // > 1 node in list, & element is not in head
             intSLLNode *pred, *tmp;
             for (pred = head, tmp = head->next;
                 tmp != nullptr && tmp->info != el;
                 pred = pred->next, tmp = tmp->next); //loop ends
             if (tmp != nullptr) { // 'tmp->info == el' is true
                   pred->next = tmp->next;
```

Member Function **deleteNode (contd.)** (13)

```
             // contd. from previous slide .
             if (tmp == tail)  tail = pred;
             delete tmp; // and delete non-head node
           } // inner else ends
    } // outer else ends
} fn ends
```

Here too, a **seg-fault is imminent** if we replace the test shown in red on previous slide with:

```
if (tmp->info == el)
```

So, be sure to **avoid that pitfall!**

Concluding Notes on the **intSLList** class

- To create an object called lst from the class intSLList:

```
intSLList lst;
```

- The SLL implementation we just saw uses two classes:
  - The list nodes created from class intSLLNode
  - The head and tail pointers from class intSLList that point to the first and last nodes of the list
- List nodes **cannot** be accessed without using head or tail pointers which are private data members of class intSLList; thus only member fns from the same class may access them
  - Since only head & tail pointers can provide access to list nodes, some textbooks designate data members as 'public' without violating information hiding principles. While this approach is fine, it is better to declare data members as 'protected'
  - If data members of intSLLNode were 'private' then classes derived from intSLLNode would not be able to access them

Asymptotic Time Complexity of SLL Operations

Asymptotic running time complexity ..

- Creation of (empty) list:
- Creation of node:
- Insertion at head of list:
- Insertion at tail of list:
- Deletion at head of list:
- Deletion at tail of list:
- Insertion or deletion in middle of list:
- Search nodes in list:
- Destruction of list:

What about average case run-times?

Asymptotic Time Complexity of SLL Operations

Asymptotic running time complexity ..

- Creation of (empty) list: O(1)
- Creation of node: O(1)
- Insertion at head of list: O(1)
- Insertion at tail of list: O(1)
- Deletion at head of list: O(1)
- Deletion at tail of list: O(n)
- Insertion or deletion in middle of list: O(n)
- Search nodes in list: O(n)
- Destruction of list: O(n)

What about average case run-times?

Doubly Linked Lists (DLLs)

We will create a DLL using a strategy that is very similar to that used in SLLs except that:

- Each node now has two pointers (prev, next) instead of one
- Therefore, the new default constructor takes three input parameters, not two
- Operations near or just before the list's tail can be done easily using the tail pointer instead of having to traverse the list from head to tail
- Data member info is declared to be of generic type T instead of int

  Thus, templates make the list flexible enough to store any kind of data, not just integers

Our SLL code can be easily modified to use templates too!

A Class for DLL Nodes (in header file dllnode.h) (1)

```
#ifndef _DLL_NODE
#define _DLL_NODE
template <typename T>
class DLLNode {  // Objects of this class are nodes in our DLL
  template<typename U>    // Since DLL class is templated
  friend class DLL;       // To allow access to protected node
                          // members from DLL class
  public:
     DLLNode(const T& el, DLLNode *n=nullptr, DLLNode *p=nullptr)
      {
        info = el;
        next = n;
        prev = p;
      }

  protected:
     T        info;            // generic type data
     DLLNode  *next, *prev;   // ptrs to next, prev nodes
};
#endif
```

Class To Manage List of DLL Nodes (in header file dllist.h) (2)

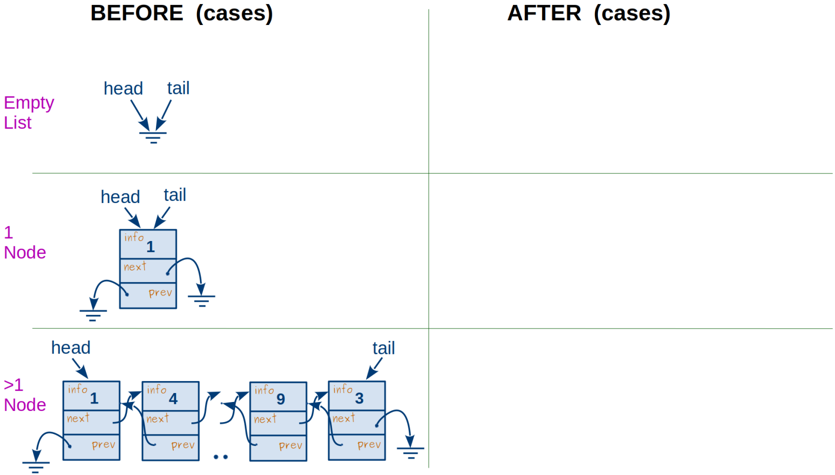
```
#ifndef _DLL
#define _DLL

template<typename T>
class DLL {       // Declaration for class DLL
     public:
             DLL() { head = tail = nullptr; }
             void addToDLLTail(const T&);
             T deleteFromDLLTail();
               :  :
     private:
             DLLNode<T> *head, *tail;  // Again, note the '<T>'
};

// Since we are using templates, member fn definitions go HERE,
// and not in a .cc file

#endif  // NOTE: 'ifndef' ends AFTER member fn  definitions
```

Strategy: The "Before" & "After" Sketches (3)



BEFORE (cases)

AFTER (cases)

Empty List

1 Node

>1 Node

Figure: Use this template to sketch your strategy

Member Function **addToTail** (4)

Sketch your strategy first; use template on previous slide

```
template<typename T>
void DLL<T>::addToDLLTail(const T& el) {
      if (tail != nullptr) {
          tail = new DLLNode<T>(el, nullptr, tail);
          tail->prev->next = tail;
      }
      else head = tail = new DLLNode<T>(el);
}
```

Member Function **deleteFromTail** (5)

Sketch your strategy first; use template on slide 30

```
template<typename T>
T  DLL<T>::deleteFromTail() {
      if (head != nullptr) {  // if non-empty list
          T el = tail->info;
          if (head == tail) {  // if list has just one node
             delete tail;
             head = tail = nullptr;
          }
          else {  // if list has >1 node
             tail = tail->prev;
             delete tail->next;
             tail->next = nullptr;
          }
          return el;
      }
      else throw(EMPTY_LIST);
}
```

Asymptotic Time Complexity of DLL Operations

Asymptotic running time complexity ..

- Creation of (empty) list:
- Creation of node:
- Insertion at head of list:
- Insertion at tail of list:
- Deletion at head of list:
- Deletion at tail of list:
- Insertion or deletion in middle of list:
- Search nodes in list:
- Destruction of list:

What about average case run-times?

Circular Lists: CSLLs & CDLLs

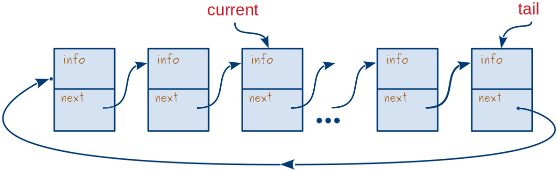


Figure: A Circular Singly Linked List (CSLL)

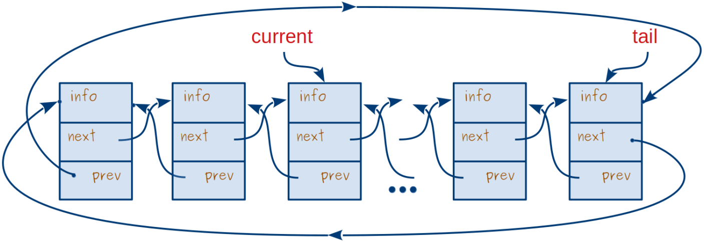


Figure: A Circular Doubly Linked List (CDLL)

Successor of the tail is always the head; **head ptr is not needed**

CSLL: Node Insertion

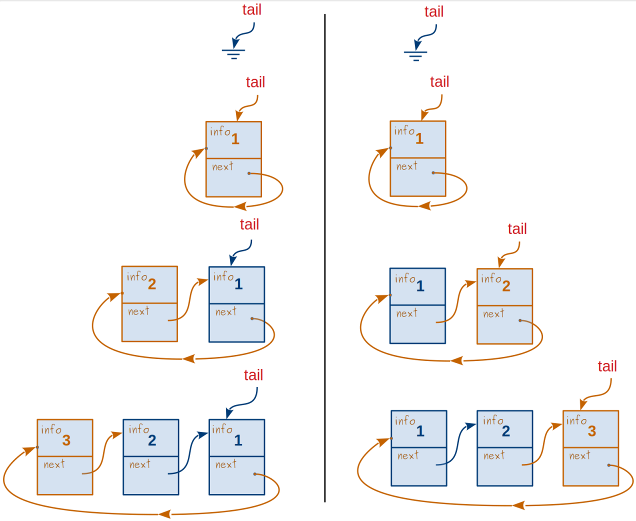


Figure: (a) Insertion at front (b) Insertion at rear

Practice!

Do the following:

Use the nodes from intSLLNode class to create a CSLL class ..

Start by writing the declaration for a CSLL class, and then rewrite all the member functions that you wrote for the intSLList class. Be sure to sketch the "before" and "after" cases prior to writing code