RC8

Linked List, Template and Container

Outline

- Linked list
- Template
- Container of pointers

Linked List

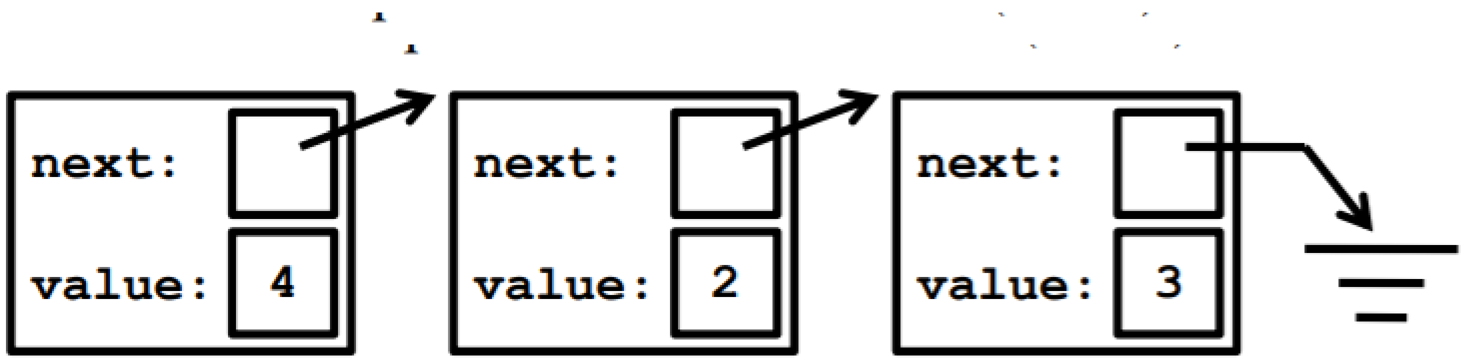
Linked List

- What is linked list?
  - An ADT, which……
  - Stores data in series
  - Is mutable (not like list_t)
  - Is expandable – we can insert or remove data (not like array)
  - Slow random access (i.e. slow to read/write a[i])

Linked List: Implementation

- Firstly, we define the unit linked list - node
  - Stores data and **address** of next node in the list
  - Question: Can we define *next* as a node instead of node*?
- next == NULL means we have reached end of list.

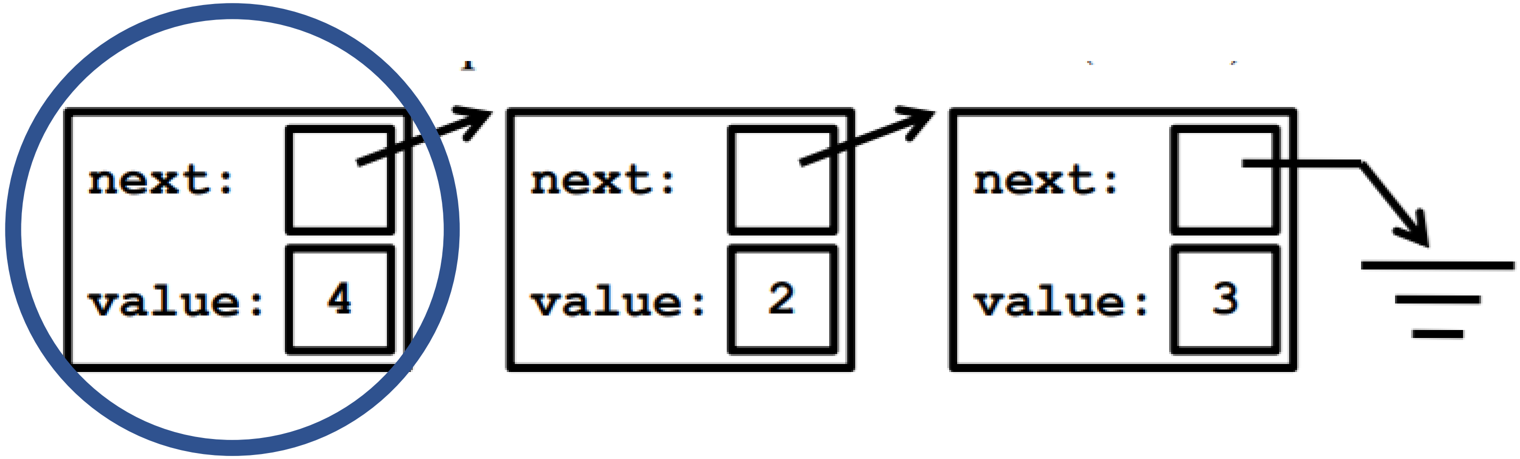
```
struct node {
    node *next;
    int value;
}
```



Linked List: Implementation

- The liked list only need one (private) data member
  - You can also add others if needed (e.g., size).
  - Once we reach the first node, we can reach any node in the list

```
class list{
    node *first;
public:
    ......
}
```



Linked List: Traversal

- There is an example of traversal of linked list in your lecture slides
  - This is one reason we define *first* as a pointer – convenience

```
int IntList::getSize() {
// Effect: return # of items in this list
    int count = 0; node *current = first;
    while(current){
        count++;
        current = current->next;
    }
    return count;
}
```

Linked List: Member functions

- bool isEmpty()
- void insert(int v) // to the beginning
  - must use new
  - modify *first*
  - boundary case
- int remove() // the first node
  - must use delete
  - modify *first*
  - boundary case

Linked List: Member functions

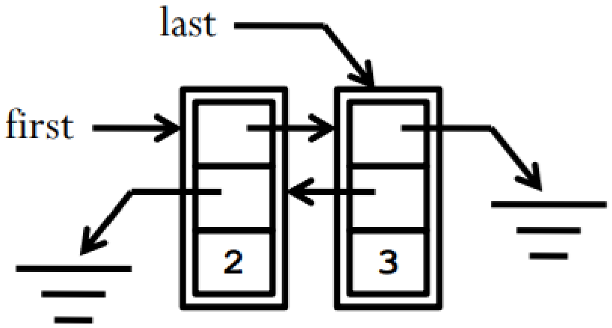
- list()
- list(const list&l)
  - deep copy – when data in *l* changes, data in *this* will not change
- ~list()
- list &operator=(const list &l)
  - clear data & deep copy
  - boundary case

Linked List: Double ended list

```
class IntList {
  node *first;
  node *last;
 public:
  ...
};
```

- Now we can insert/remove at the other end.
- Which of member functions need to be modified?
- However, remove from last is too slow
  - need to traverse from first to the "second last" node
- So double linked list is needed
  - example from lecture slides: list (2, 3)

```
struct node {
  node *next;
  node *prev;
  int   value;
};
```



Template

Template

- Why do we need template?
  - We can define class/function for different type.
  - Write more reusable code – polymorphism.
- Example: linked list of integer and linked list of string

Template: How to write

- Declaration: add template<class T> before declaration

```
template<class T>
class List {
    ......
}
```

```
template<class T>
T max(T x, T y);
```

- Definition:
  - normal function:

```
template<class T>
T max(T x, T y) {
    ......
}
```

  - class member function:

```
template<class T>
T List<T>::insert(T v) {
    ......
}
```

Template: How to write

- When use:
  - Create an instance of template class:
  - Call template function:

```
List<double> list1;
```

```
string a, b;
……
string c =
max<string>(a, b)
```

Template: Exercise

- Define a template class named CannonK with two template parameters T, U
- Write its default constructor header
- Write the header its copy constructor
- Write its the header of its destructor
- Write the header of “=“ overloading
- Define a variable named airport that is a vector of CannonK with string and int

Container of Pointers

Container of Pointers

- Why do we need container of pointers?
  - Sometimes we do not want to copy the data, especially when the data is large – sometimes we are guaranteed we do not need to copy.
- Although we can easily get a container of pointers with the help of template, we do **NOT** write like this:
  - ~~list<BigThing*> ls;~~
- Instead, we define a templated container of pointers and write like this:
  - list_of_pointer<BigThing> ls;
- Difference in implementation of two templated container is in the next page

Container of Pointers

```
template <class T>
class List {
  public:
    ...
    void insert(T v);
    T     remove();
  private:
    struct node {
        node *next;
        T     o;
    };
    ....
};
```

```
template <class T>
class List {
  public:
    ...
    void insert(T *v);
    T     *remove();
  private:
    struct node {
        node *next;
        T     *o;
    };
    ....
};
```

Container of Pointers: 1 invariant & 3 rules

- At-most-once invariant: any object can be **linked** to at most one container at any time through pointer.
- Existence: An object must be dynamically allocated before a pointer to it is **inserted**.
- Ownership: Once a pointer to an object is **inserted**, that object becomes the property of the container. No one else may use or modify it in any way.
- Conservation: When a pointer is **removed** from a container, either the pointer must be inserted into some container, or its reference must be deleted.
- Go to lecture slides for example!

Reference

- VE280-2021FA Lecture slides, Weikang Qian.
- VE280-2021SU Final RC, Jiayao Wu