IM 1003: Computer Programming
Data Structures

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Outline

- **Basic ideas**
- Lists: **`class JobList`**
- Linked lists: **`JobLinkedList`**
- More data structures
- Topics not covered in this semester

Data structures

- A **data structure** is a specific way to **store** data.
- Usually it also provides interfaces for people to **access** data.
- Real-life examples: A dictionary.
  - It stores words.
  - It sorts words alphabetically.

Data structures

- In large-scale software systems, there are a lot of data. We want to create data structures to store and manage them.
- We want our data structures to be **safe**:
  - People can access data only through managed interfaces.
  - Think about encapsulation!
- We need them to be **effective**:
  - We can store and access data correctly.
- We also want them to be **efficient**:
  - Operations can be done in a short time.
  - Consider a dictionary with words not sorted!

Data structures

- An array is a very simple data structure.
- Is it safe, effective, and efficient?
  - Safety: Only if suitable interfaces are provided.
  - Effectiveness: Only if suitable interfaces are provided.
  - Efficiency: To be discussed later.
- Therefore, our first attempt will be to build a "more complicated" data structure based on an array.

Outline

- Basic ideas
- **Lists: `class JobList`**
- Linked lists: **`JobLinkedList`**
- More data structures
- Topics not covered in this semester

Lists

- A list is a **linear** data structure. It stores items in a line.
  - E.g., a dictionary, a contact list, a personal schedule, etc.
- As an example, we will implement a job list, which stores jobs.
- The class **`JobList`** will use an **array** to store jobs.
  - Jobs with a smaller index has higher priority.
- More importantly, it will provide **interfaces** to access those jobs.
  - The array will be a **private** or **protected** member variable.
  - The interfaces will be **public** member functions.

Job

```
class Job
{ // nothing special
private:
  string name;
  int hour;
public:
  Job() { this->name = ""; this->hour = 0; }
  Job(string name, int hour)
  { this->name = name; this->hour = hour; }
  void setHour(int hour) { this->hour = hour; }
  string getName() { return this->name; }
  double getHour() { return this->hour; }
  void print() {
    cout << "(" << this->name
         << ", " << this->hour << ")";
  }
};
```

JobList

```
const int MAX_JOBS = 100; // a global variable

class JobList
{
private:
  Job jobs[MAX_JOBS]; // where we store the data
  int count; // other attributes
public:
  JobList() { this->count = 0; }
  // interfaces
  int getCount() { return this->count; }
  bool insert(Job job, int index);
  Job remove(int index);
  void print();
};
```


JobList::print()

```
void JobList::print()
{
  for(int i = 0; i < this->count; i++)
  {
    cout << "Job " << i + 1 << ": ";
    this->jobs[i].print();
    cout << endl;
  }
}
```


JobList::insert()

```
bool JobList::insert(Job job, int index)
{
  if(index < 0 || this->count == MAX_JOBS)
    return false; // fail to insert
  else if(index > this->count) // insert at the end
    this->jobs[this->count] = job;
  else // usual insertion
  {
    for(int i = this->count - 1; i >= index; i--)
        this->jobs[i+1] = this->jobs[i];
    this->jobs[index] = job;
  }
  this->count++;
  return true;
}
```


JobList::remove()

```
Job JobList::remove(int index)
{
  Job toRemove; // to be removed and returned
  if(index < 0 || this->count == 0)
    return toRemove; // nothing to remove
  else if(index > this->count) // remove the last one
    toRemove = this->jobs[this->count];
  else // usual removal
  {
    toRemove = this->jobs[index];
    for(int i = index; i < this->count - 1; i++)
      this->jobs[i] = this->jobs[i+1];
  }
  this->count--; // the effective action of removal
  return toRemove;
}
```

Remarks

- Is **JobList** safe, effective, and efficient?
  - Safety: People can access these data **only through** public interfaces.
  - Effectiveness: We have implemented **fail-safe** interfaces.
  - Efficiency: Not so efficient! Insertion and removal may need to move all jobs (i.e., $O(n)$).
- Drawbacks:
  - There is a limit on the total number of jobs.
  - A lot of storage spaces are wasted.
- These drawbacks exist for almost every data structure implemented with arrays, even with dynamic memory allocation.
- We will introduce another "list" that does not use an array.

Outline

- Basic ideas
- Lists: **class JobList**
- **Linked lists: JobLinkedList**
- More data structures
- Topics not covered in this semester

Linked lists

- A **linked list** is a list implemented by using **pointers** so that "each element has a pointer pointing to the next element".
- Advantages:
  - No limit on the number of elements stored.
  - Dynamically allocate memory spaces. Can save spaces.
  - Efficiency may be improved (in some cases).
- Disadvantages:
  - Harder to implement.
  - Efficiency may be worsen (in some cases).

Job

```
class Job
{
friend class JobLinkedList; // discussed later
private:
  string name;
  int hour;
  Job* next; // the pointer pointing to the next job
public:
  Job()
  {
    this->name = "";
    this->hour = 0;
    this->next = NULL;
    // has the next job only if put in a list
  }
  // (continue to the next slide)
```

Job

```
// (continue from the previous slide)
Job(string name, int hour)
{
  this->name = name;
  this->hour = hour;
  this->next = NULL;
}
void setHour(int hour) { this->hour = hour; }
string getName() { return this->name; }
double getHour() { return this->hour; }
void print();
{
  cout << "(" << this->name
       << ", " << this->hour << ")";
}
};
```


JobLinkedList

```
class JobLinkedList
{
protected:
  int count;
  Job* head; // pointing to the first Job
public:
  JobLinkedList() {
    this->count = 0;
    this->head = NULL;
  }
  ~JobLinkedList(); // for dynamically-allocated space
  // same interfaces
  int getCount() { return this->count; }
  bool insert(Job job, int index);
  Job remove(int index);
  void print();
};
```


JobLinkedList::print()

```
void JobLinkedList::print()
{
  Job* temp = this->head;
  for(int i = 0; i < this->count; i++)
  {
    cout << "Job " << i + 1 << ": ";
    temp->print();
    cout << endl;
    temp = temp->next; // move to the next job
  }
}
```


JobLinkedList::insert()

```
bool JobLinkedList::insert(Job job, int index)
{
  Job* toInsert = new Job(job.name, job.hour);
  if(index < 0) // fail-safe
    return false;
  else if(index == 0) // insert it as the head
  {
    if(this->count > 0)
      toInsert->next = this->head;
    this->head = toInsert;
  }
  // (continue to the next slide)
```

JobLinkedList::insert()

```
  // (continue from the previous slide)
  else // insert it somewhere in the list
  {
    if(index > this->count) // fail-safe
      index = this->count;
    Job* temp = this->head; // find the place
    for(int i = 0; i < index - 1; i++)
      temp = temp->next;
    toInsert->next = temp->next; // insertion
    temp->next = toInsert;
  }
  this->count++;
  return true;
}
```


JobLinkedList::remove()

```
Job JobLinkedList::remove(int index)
{
  Job toRemove;
  if(index < 0 || this->count == 0)
    return toRemove; // return an empty job
  else if(index <= 1)
  {
    toRemove = *(this->head); // return the head
    Job* temp = this->head; // removal
    this->head = temp->next;
    delete temp;
  }
  // (continue to the next slide)
```


JobLinkedList::remove()

```
  // (continue from the previous slide)
  else
  {
    Job* temp = head; // find the place
    for(int i = 0; i < index - 2; i++)
      temp = temp->next;
    Job* tempNext = temp->next; // removal
    temp->next = tempNext->next;
    toRemove = *tempNext; // return the removed one
    delete tempNext;
  }
  this->count--;
  toRemove.next = NULL;
  return toRemove;
}
```


Common errors

- If a **Job** pointer **job** is **NULL**, then accessing **job->next** generates a run-time error.
- Forgetting to set **next** to **NULL** may also create run-time errors.

Remarks

- In general, a list is a linear data structure. It stores multiple "nodes", which is another elementary data structure.
- When an **`A`** "has a" **`B`**, usually we make **`A`** as **`B`**'s friend.
  - A job linked list has a job.
- In a linked list, each node contains a pointer for the next node.
- For our **`JobLinkedList`**:
  - There is no limit on the number of nodes stored.
  - Spaces are saved by using dynamic memory allocation.
  - Efficiency is roughly the same as **`JobList`**: Insertion and removal are $O(n)$.

Encapsulation

- We implemented two lists:
  - **`JobList`**: using an array.
  - **`JobLinkedList`**: using pointers.
- Though the private storages are different, the **public interfaces** are identical!

```
JobLinkedList();
int getCount();
bool insert(Job job, int index);
Job remove(int index);
void print();
```

  - One **uses** these two classes in the same way.
  - Except for **`JobList`** there is a limit on the number of jobs.

Encapsulation

- One does not need to (also should not) know the list is created.
- One should just know **how to use it**.
- What if I can see and access the array in **`JobList`**?
  - I may write codes to access the array directly: The data structure is not safe.
  - In the future if the implementation of **`JobList`** is modified, I may also need to modify my codes even if the interfaces all remain the same.

Destructors

- If dynamic memory allocation is implemented, we need to release those dynamically-allocated spaces by the delete statement.
- Consider the following main function in the next slide.

Linked lists: Destructors

```
int main()
{
  JobLinkedList jll;
  Job j1("j1", 1), j2("j2", 2), j3("j3", 3);
  // memory spaces are allocated statically
  jll.insert(j1);
  jll.insert(j2);
  jll.insert(j3);
  // 3 new statements are executed
  return 0;
} // no delete statement is executed!
  // a destructor is useful in this case
```


JobLinkedList::~JobLinkedList()

```
JobLinkedList::~JobLinkedList() // version 1
{
  Job* temp = this->head;
  Job* tempNext = NULL;
  // Do not write "Job* tempNext = this->head->next;"
  // If we do so, what happens on an empty list?

  for(int i = 0; i < this->count; i++)
  {
    tempNext = temp->next;
    delete temp; // release memory
    temp = tempNext;
  }
}
```


JobLinkedList::~JobLinkedList()

```
JobLinkedList::~JobLinkedList() // version 2
{
  while(this->count > 0)
    this->remove(0); // release memory
  // do not use
  // for(int i = 0; i < this->count; i++)
  //   this->remove(0);
  // why?
}
```


Good Programming style

- Be very careful when using pointers.
- Write your codes slowly and as clear as possible.
  - Compile and test your program whenever you complete a function!
- When there is a run-time error, check whether you are accessing a `NULL` pointer.
- Check whether you need a destructor or a copy constructor when your class has a pointer member.

Outline

- Basic ideas
- Lists: **`class JobList`**
- Linked lists: **`JobLinkedList`**
- **More data structures**
- Topics not covered in this semester

Stacks and queues

- A **stack** is a special list. A **queue** is another special list.
- Nodes can not be inserted/removed at any place we want.
  - Stack: last-in-first-out (**LIFO**). A node can only be inserted and removed at the **top** of the stack.
  - Queue: first-in-first-out (**FIFO**). A node can only be inserted at the **tail** and removed at the **head**.
- Many real-life situations can be modeled as stacks or queues.

Applications of stacks

- The poker game solitaire.
- The Hanoi tower.
- Function calls in your programs.
- Graph traversal: Depth-first search (DFS).
- Calculators.

Applications of queues

- Consumer waiting lines.
- FIFO job scheduling.
- Topological sorting.
- Graph traversal: Breadth-first search (BFS).

Creating a job stack by inheritance

- Though not realistic, we will implement a job stack by inheriting the job linked list.
  - The implementation of a job queue is left to you.
- This example shows
  - The application of **inheritance**: Once you have a list, it is very easy to create a stack or a queue.
  - The application of **encapsulation**: The idea of interfaces.
  - The application of **protected inheritance**: Not all public members of the parent class should be public for the child class.

JobStack

```
class JobStack : protected JobLinkedList
{
public:
  JobStack();
  ~JobStack();
  void push(Job job);
  Job pop();
  void print();
};
/* protected: we want to hide insert() and remove() inherited from
   JobLinkedList */
```


JobStack

```
JobStack::JobStack() : JobLinkedList()
{
  ;
}

JobStack::~JobStack()
{
  ;
}

void JobStack::print()
{
  JobLinkedList::print();
}
```


JobStack

```
// insert at top (end)
void JobStack::push(Job job)
{
  JobLinkedList::insert(job, this->count);
}

// remove the one at top (end)
Job JobStack::pop()
{
  return JobLinkedList::remove(this->count);
}
```

Remarks

- The class **JobStack** is indeed a stack. It is safe and effective.
- However, it is not the most efficient implementation.
  - Operations are executed through another class.
  - **push()** and **pop()** are both $O(n)$. With a **Job** pointer **tail**, they can be both $O(1)$ (the codes will be more complicated).
- Be careful that **insert()** and **remove()** should be hided.
  - If you use public inheritance, you may override them.
- Inheriting **JobList** also creates a safe and effective job stack.

Trees

- A list, stack, or queue is a linear (one-dimensional) data structure.
- A tree is a two-dimensional data structure.
- A binary tree is a useful two-dimensional data structure.

```
class BTNode
{
private;
  BTNode* left;
  BTNode* right;
  // …
}
```


Outline

- Basic ideas
- Lists: **class JobList**
- Linked lists: **JobLinkedList**
- More data structures
- **Topics not covered in this semester**

Topics not covered in this course

- Operator overloading.
  - **JobLinkedList aList; aList[3];**
- Template classes and functions.
  - Job list, product list, consumer list, …
- And many others.

Topics not covered in this course

- Even if you want to develop a system only with those skills you learned, there are too many details to cover.
- Also, experience is required to write programs efficiently and correctly.
- Learning is necessary but not sufficient. Practicing is also necessary.
- Practice makes perfect!
  - This is also true for the instructor of this course.

Finish!! Thank you~~~~