Programming Design

Data Structures

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Outline

- **Basic ideas**
- Lists: **`class JobList`**
- Linked lists: **`JobLinkedList`**
- More data structures

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**, **effective**, and **efficient**
  - Encapsulation: People can access data only through managed interfaces.
  - We can store and access data correctly.
  - The number of steps required for a task is small; 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

Lists

- A list is a **linear** data structure. It stores items in a line.
  - E.g., a dictionary, a personal schedule, a team of characters, 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();
  Job(string name, int hour);
  void print();
};
```

```
Job::Job() : name(""), hour(0)
{
}

Job::Job(string name, int hour)
  : name(name), hour(hour)
{
}

void Job::print()
{
  cout << "(" << this->name
       << ", " << this->hour << ")";
}
```

JobList

```
const int MAX_JOBS = 100;

class JobList
{
private:
  Job jobs[MAX_JOBS];
  int count;
public:
  JobList();
  // interfaces
  int getCount();
  void print();
  bool insert(Job job, int index);
  bool remove(int index);
};
```

```
JobList::JobList() : count(0)
{
}
int JobList::getCount()
{
  return this->count;
}
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) // fail to insert
    return false;
  else if(index > this->count) // fail to insert
    return false;
  else // usual insertion
  {
    for(int i = count - 1; i >= index; i--)
      this->jobs[i+1] = this->jobs[i];
    this->jobs[index] = job;
  }
  this->count++;
  return true;
}
```

JobList::remove()

```
bool JobList::remove(int index)
{
  if(index < 0 || this->count == 0 || index > this->count)
    return false; // nothing to remove
  else // usual removal
  {
    for(int i = index; i < this->count - 1; i++)
      this->jobs[i] = this->jobs[i+1];
    this->count--; // the effective action of removal
    return true;
  }
}
```

A main function

```
int main()
{
  JobList l;
  Job j1("Programming", 10);
  Job j2("Accounting", 20);
  l.insert(j1, 0);
  l.insert(j2, 1);
  l.print();
  l.remove(0);
  l.print();

  return 0;
}
```

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

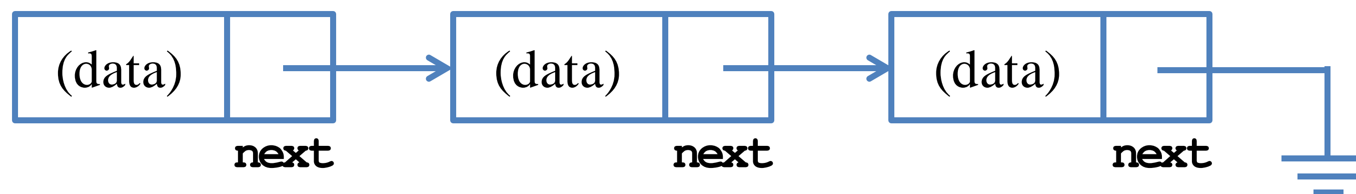
- Basic ideas
- Lists: `class JobList`
- **Linked lists: `JobLinkedList`**
- More data structures

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).

Linked lists

- Each "node" in a linked list contains the data and **a pointer**.
  - That pointer indicates the **address of the next node**.
- Conceptually:



Job (a new definition)

```
class Job
{
friend class JobLinkedList; // Why?
private:
  string name;
  int hour;
  Job* next; // the pointer
public:
  Job();
  Job(string name, int hour);
  void print();
};
```

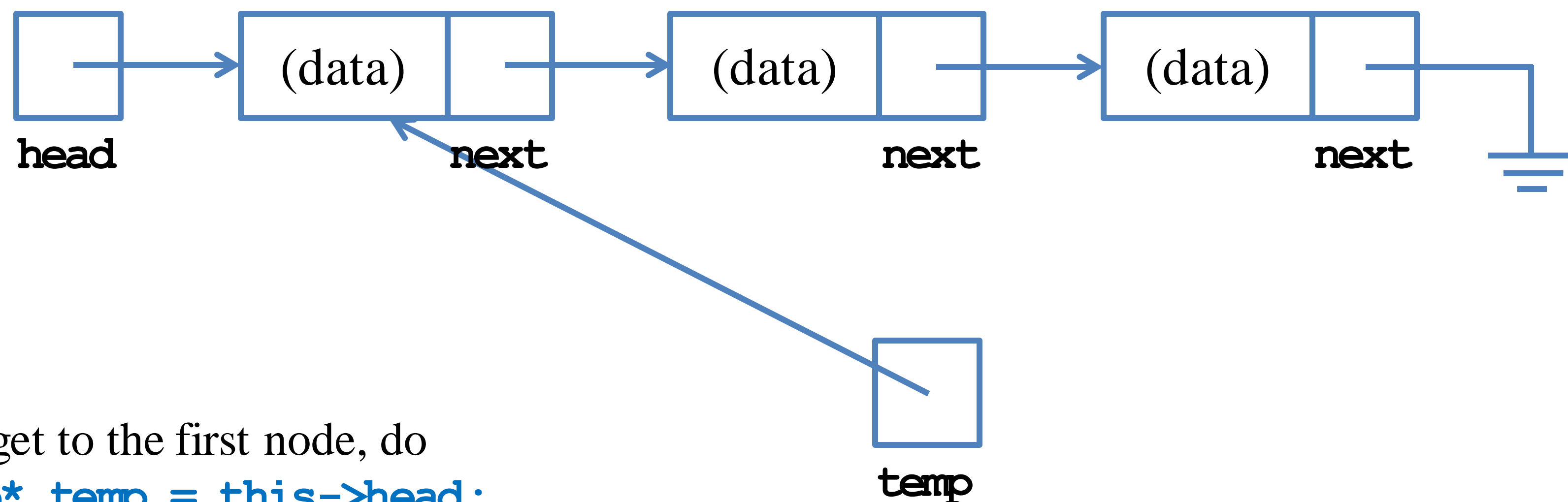
```
Job::Job()
  : name(""), hour(0), next(nullptr)
{
}

Job::Job(string name, int hour)
  : name(name), hour(hour), next(nullptr)
{
}

void Job:: print()
{
    cout << "(" << this->name
         << ", " << this->hour << ")";
}
```

JobLinkedList

```
class JobLinkedList
{
protected:
  int count;
  Job* head; // address of the first Job
public:
  JobLinkedList();
  // same interfaces
  int getCount();
  bool insert(Job job, int index);
  bool remove(int index);
  void print();
};
```

```
JobLinkedList::JobLinkedList()
  : count(0), head(nullptr)
{
}

int JobLinkedList::getCount()
{
  return this->count;
}
```

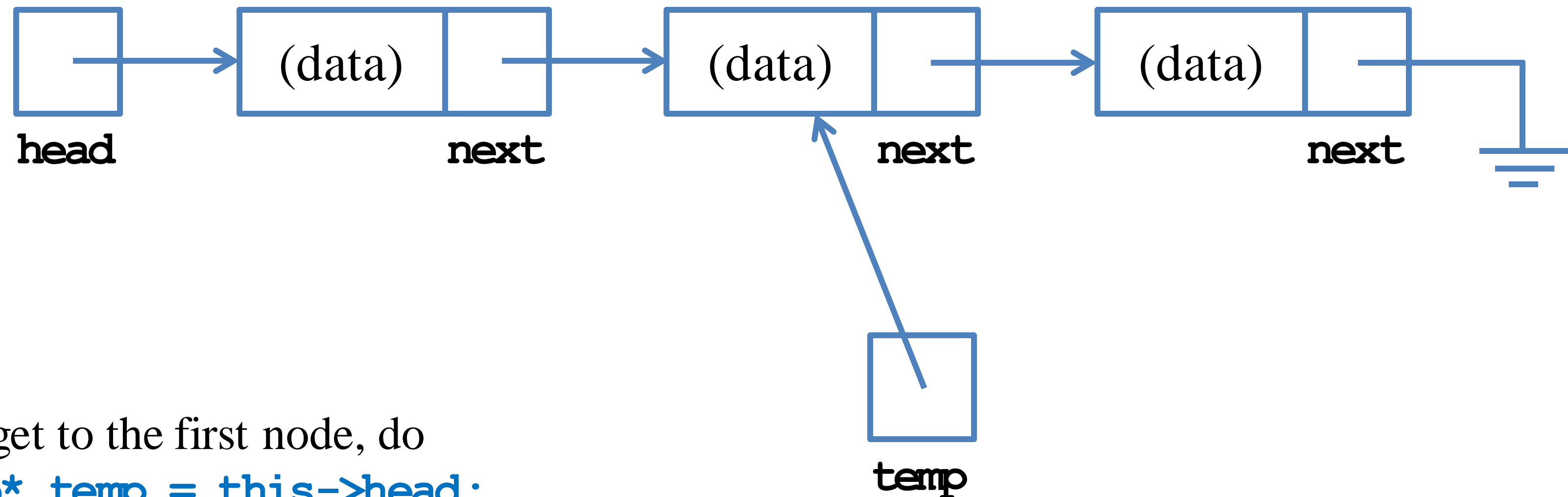
JobLinkedList::print()



To get to the first node, do

```
Job* temp = this->head;
```

JobLinkedList::print()



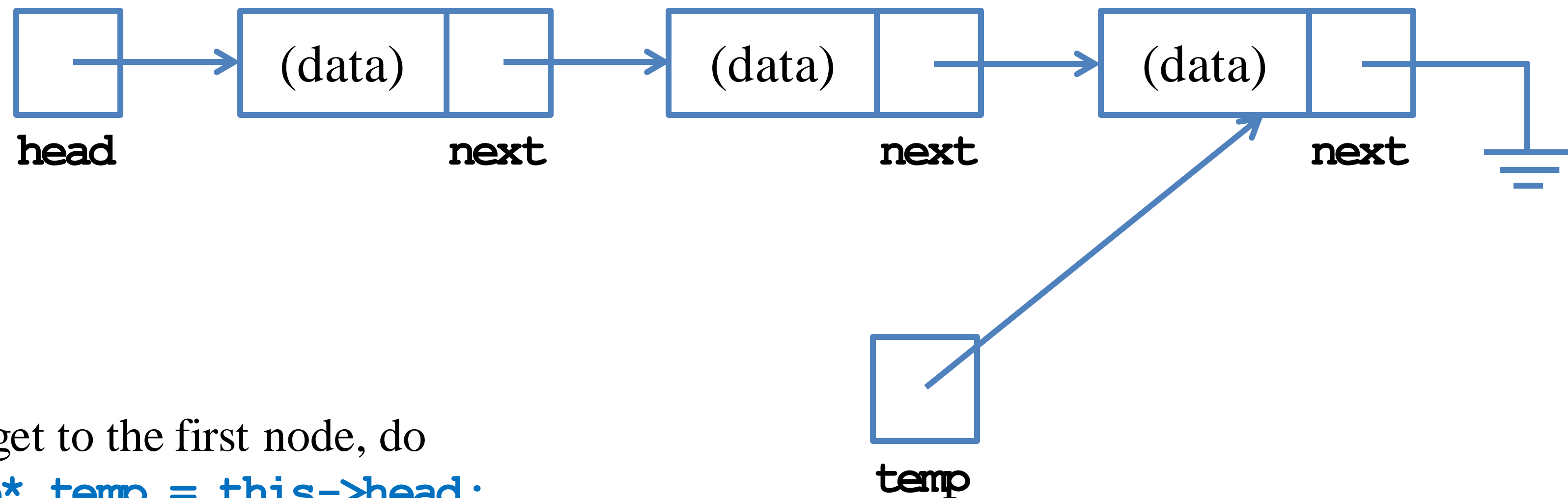
To get to the first node, do

```
Job* temp = this->head;
```

To get to the next node, do

```
temp = temp->next;
```

JobLinkedList::print()



To get to the first node, do

```
Job* temp = this->head;
```

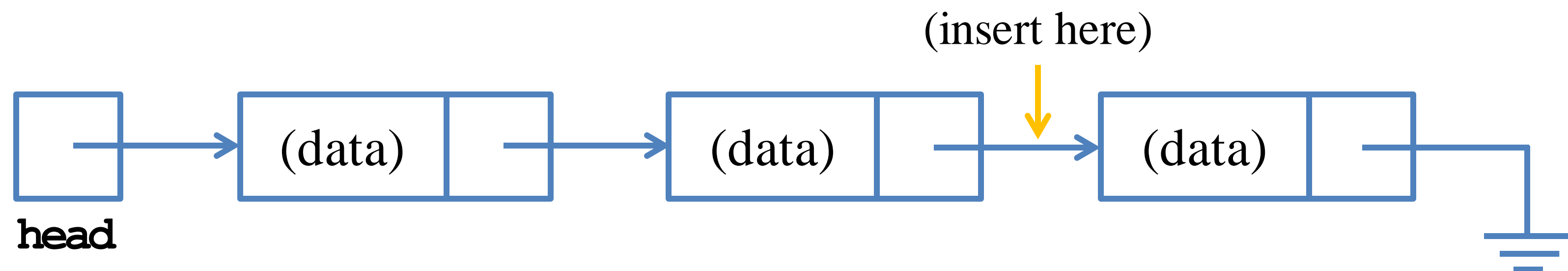
To get to the next node, do

```
temp = temp->next;
```

JobLinkedList::print()

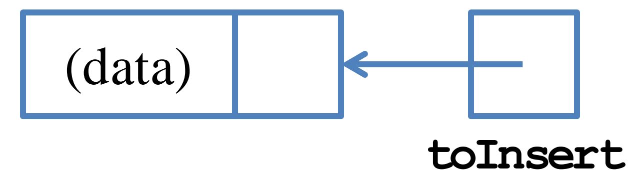
```
void JobLinkedList::print()
{
  Job* temp = this->head;
  for(int i = 0; i < this->count; i++)
  {
    cout << "Job " << i + 1 << ": "; // print out one job
    temp->print();
    cout << endl;

    temp = temp->next; // move to the next job
  }
}
```

JobLinkedList::insert()

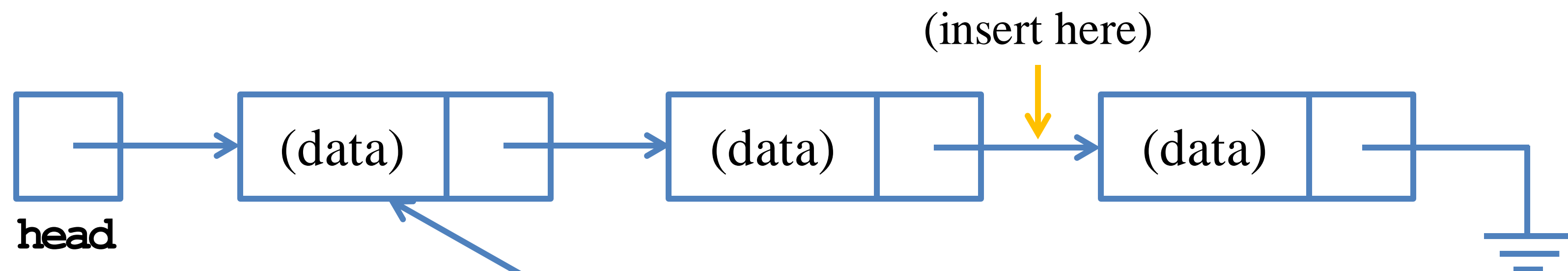


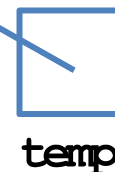
To create a node, do

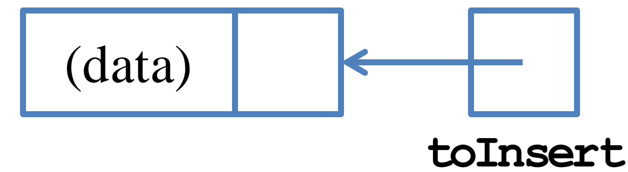
```
Job* toInsert =
  new Job(job.name, job.hour);
```



JobLinkedList::insert()



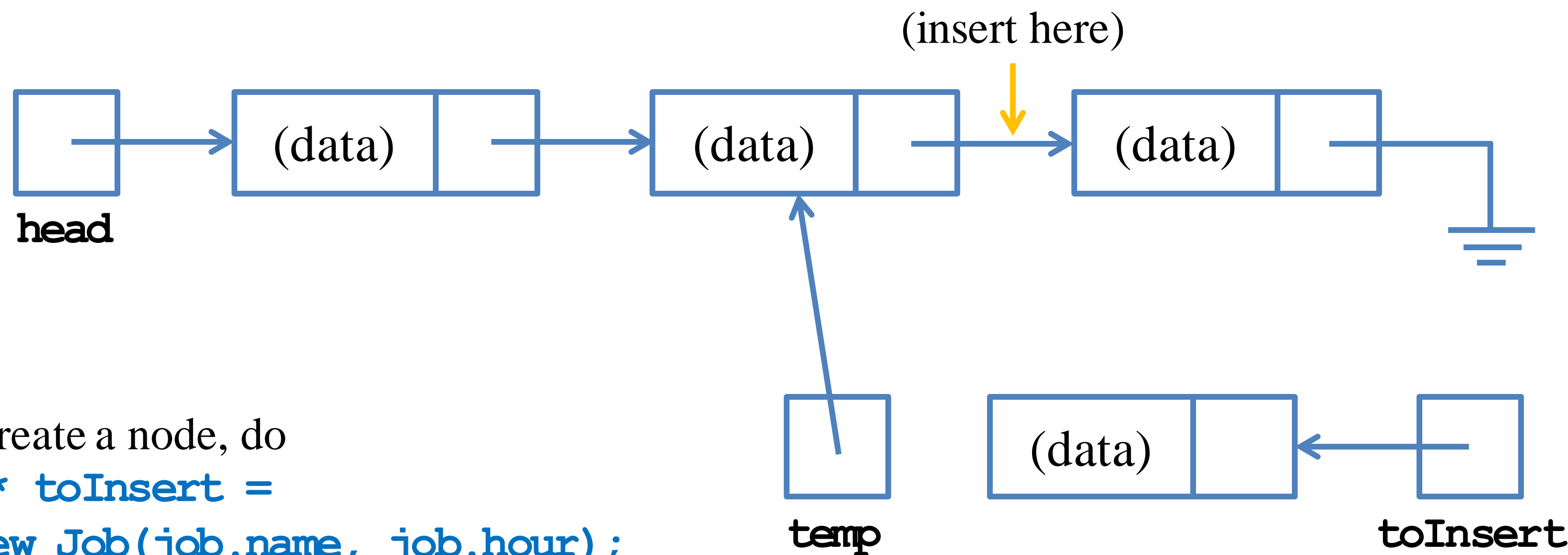




To create a node, do

```
Job* toInsert =
  new Job(job.name, job.hour);
```

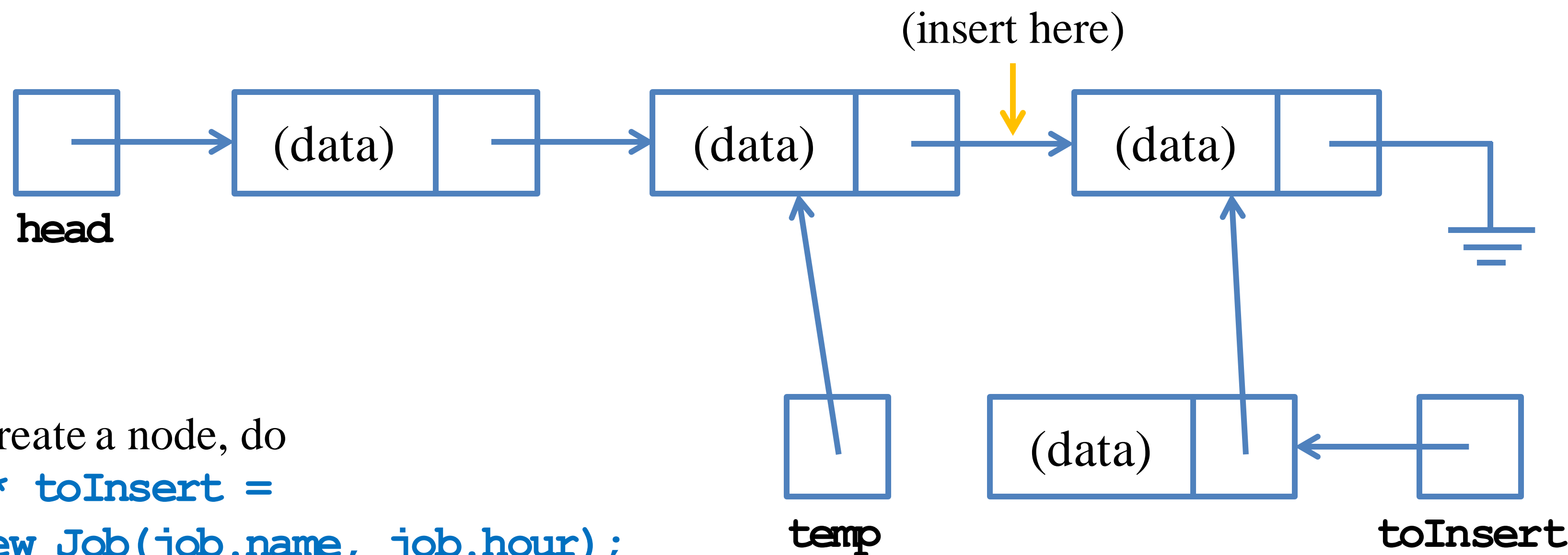
JobLinkedList::insert()



To create a node, do

```
Job* toInsert =
  new Job(job.name, job.hour);
```

JobLinkedList::insert()



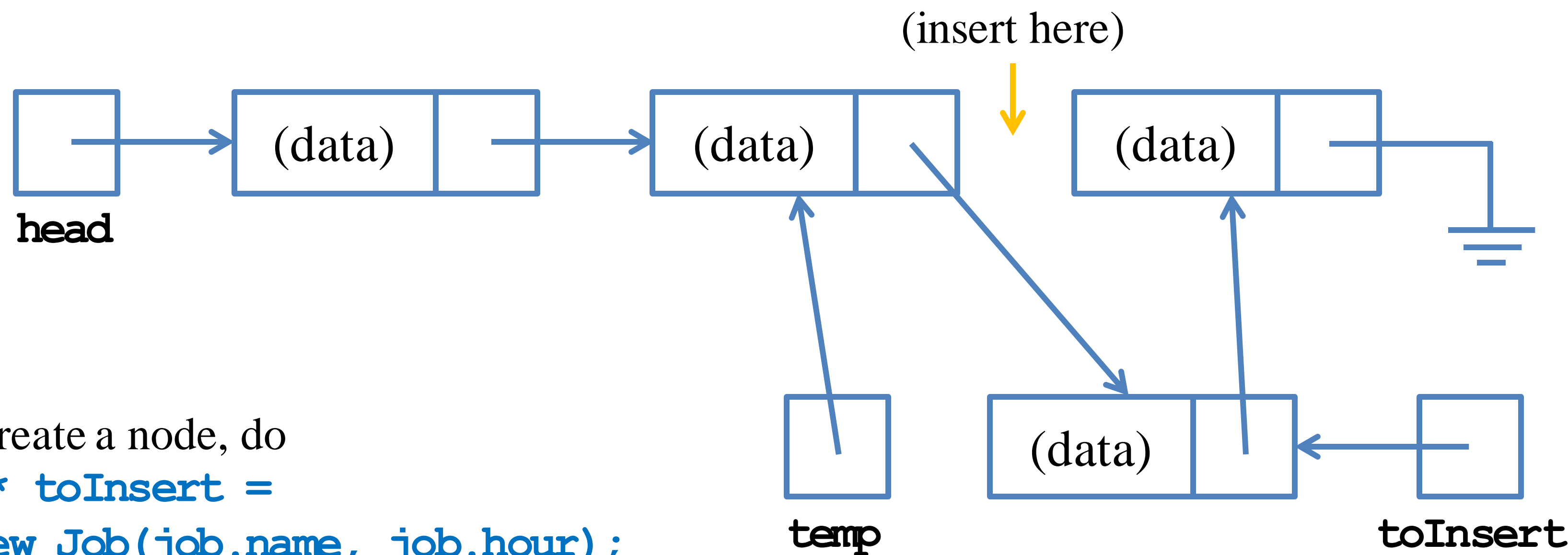
To create a node, do

```
Job* toInsert =
  new Job(job.name, job.hour);
```

To insert the node, do

```
toInsert->next = temp->next;
```

JobLinkedList::insert()



To create a node, do

```
Job* toInsert =
  new Job(job.name, job.hour);
```

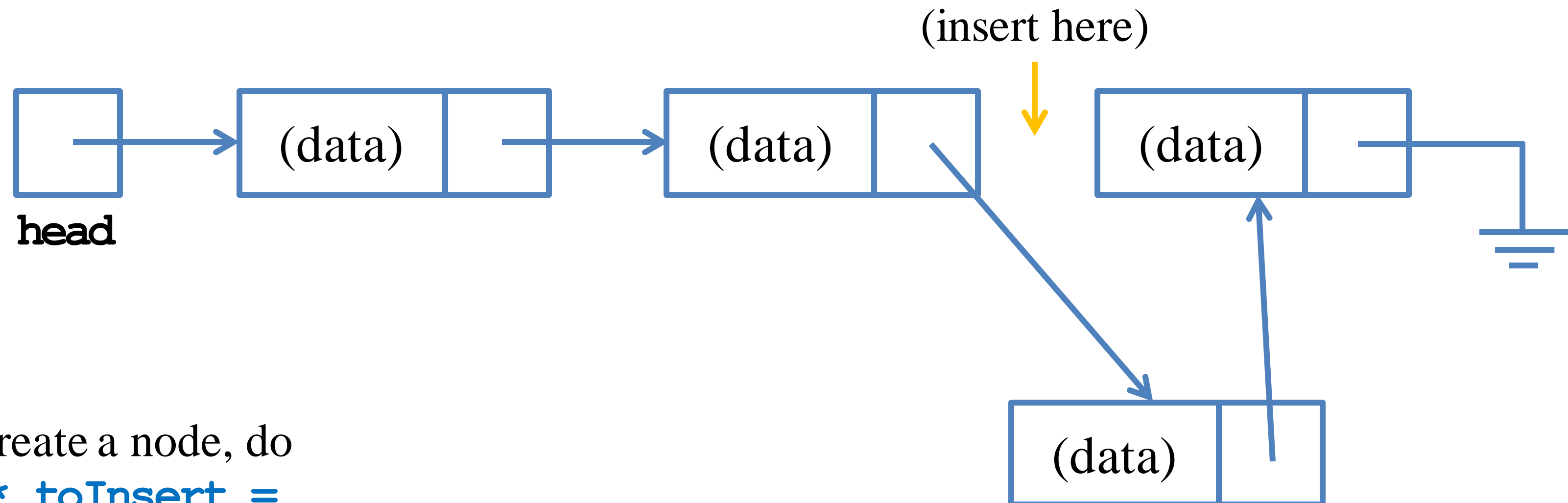
To insert the node, do

```
toInsert->next = temp->next;
```

and then

```
temp->next = toInsert;
```

JobLinkedList::insert()



To create a node, do

```
Job* toInsert =
  new Job(job.name, job.hour);
```

To insert the node, do

```
toInsert->next = temp->next;
```

and then

```
temp->next = toInsert;
```

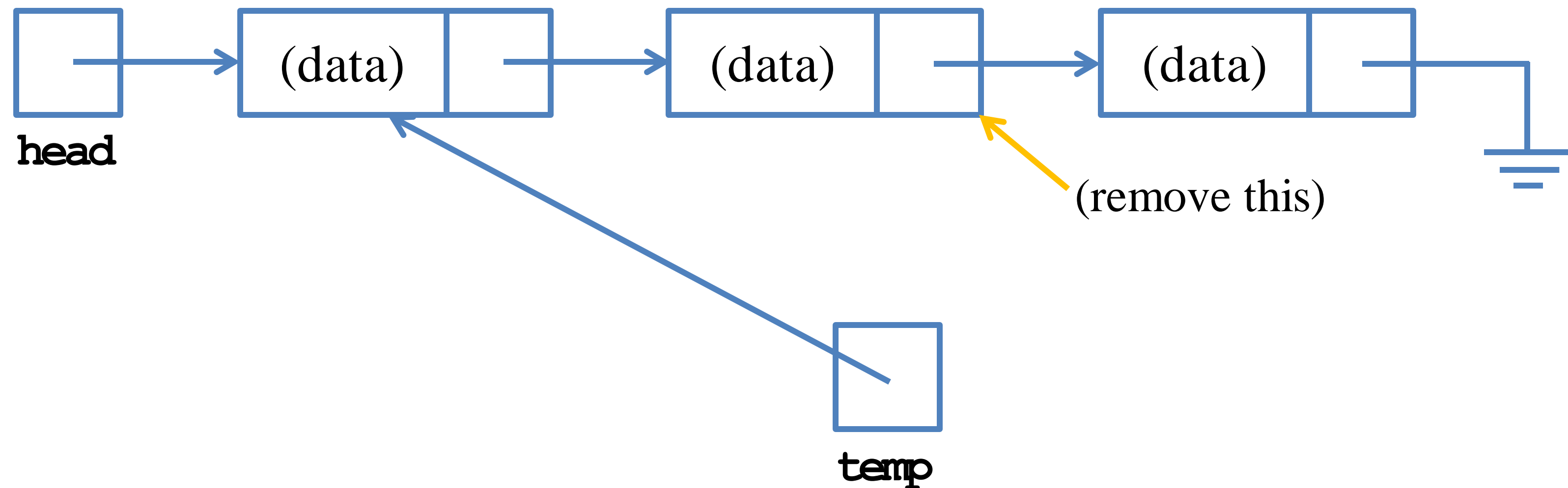
JobLinkedList::insert()

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

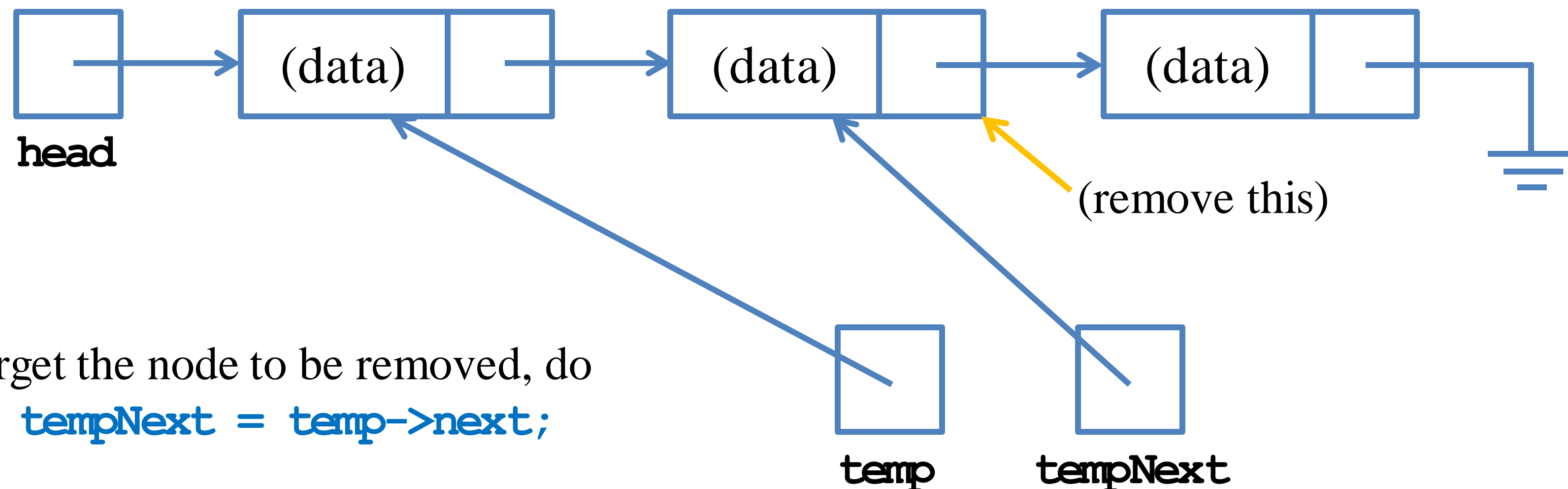
JobLinkedList::insert()

```
  // continue from the previous page
  else // insert it somewhere in the list
  {
    if(index > this->count) // fail-safe
      return false;
    Job* toInsert = new Job(job.name, job.hour);
    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()



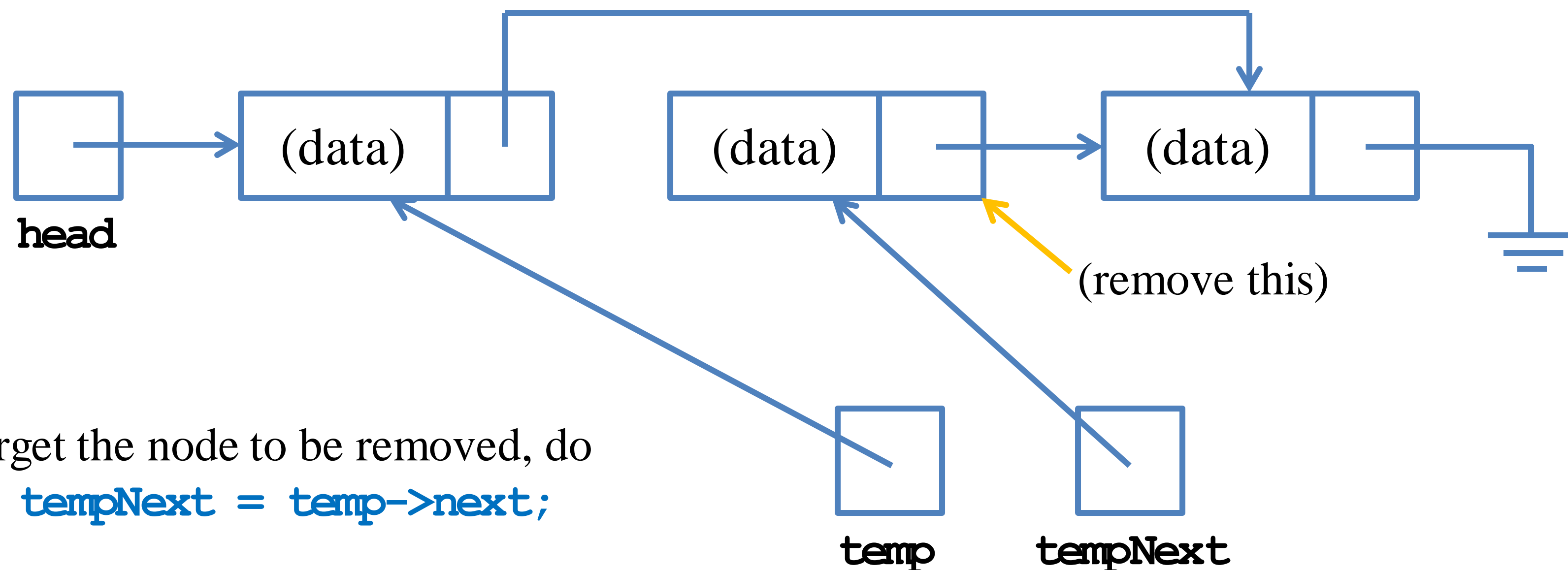
JobLinkedList::remove()



To target the node to be removed, do

```
Job* tempNext = temp->next;
```

JobLinkedList::remove()



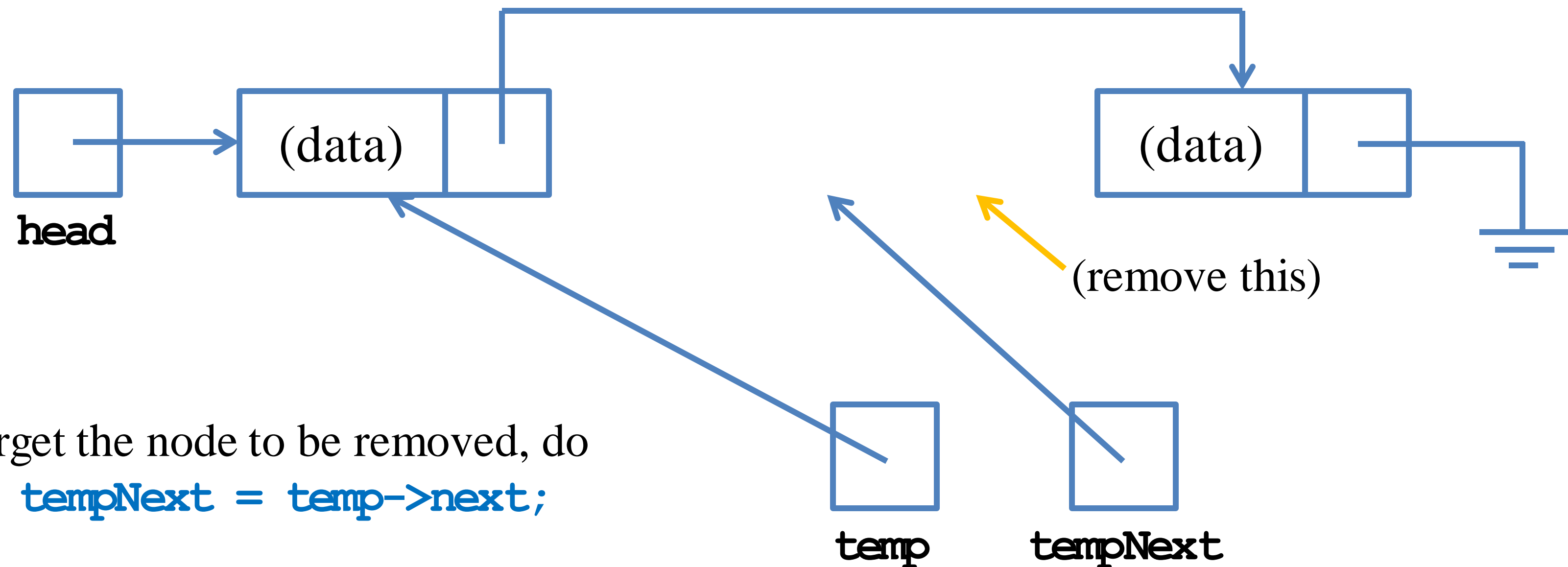
To target the node to be removed, do

```
Job* tempNext = temp->next;
```

To remove the node, do

```
temp->next = tempNext->next;
```

JobLinkedList::remove()



To target the node to be removed, do

```
Job* tempNext = temp->next;
```

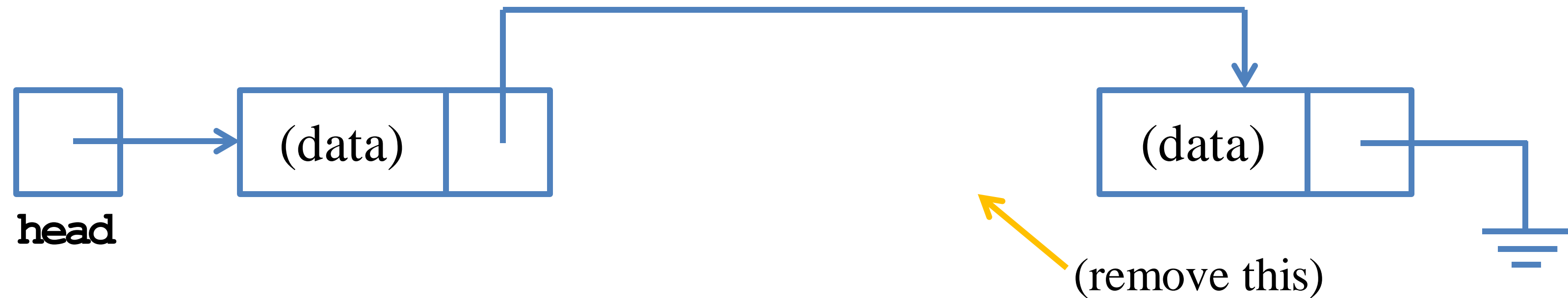
To remove the node, do

```
temp->next = tempNext->next;
```

and then

```
delete tempNext;
```

JobLinkedList::remove()



To target the node to be removed, do

```
Job* tempNext = temp->next;
```

To remove the node, do

```
temp->next = tempNext->next;
```

and then

```
delete tempNext;
```

JobLinkedList::remove()

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

JobLinkedList::remove()

```
  // continued from the previous page
  else
  {
    Job* temp = this->head; // find the place
    for(int i = 0; i < index - 2; i++)
      temp = temp->next;
    Job* tempNext = temp->next; // removal
    temp->next = tempNext->next;
    delete tempNext;
  }
  this->count--;
  return true;
}
```

Remarks

- If a **`Job`** pointer **`job`** is **`nullptr`**, then accessing **`job->next`** generates a run-time error.
  - Set **`next`** to **`nullptr`** to "create" run-time errors.
- In general, a list is a **linear data structure**. It stores multiple "nodes," where a node is another elementary data structure.
  - In a linked list, each node contains **a pointer for the next node**.
  - As a job linked list "**has a**" job, we make **`JobLinkedList`** as **`Job`**'s **friend**.
- 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 $O(n)$.

Remarks

- In general, a list stores multiple "nodes."
  - In this application, nodes are jobs.
  - In other application, nodes may be cars, students, laptops, accounts, etc.
  - **Templates** may help!
- Insertion and removal may fail.
  - Returning false to indicate a failure is fine. However, it does not enforce the caller to respond to a failure.
  - **Exception handling** may help!

Encapsulation

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

```
JobLinkedList(); // or JobList();
int getCount();
bool insert(Job job, int index);
Bool 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 how the list is implemented.
- 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).

The destructor

- If **dynamic memory allocation** is implemented, we need to release those dynamically-allocated spaces by the delete statement.
- Consider this main function:
- Let's implement a destructor.

```
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 = nullptr;
  // 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
}
```

```
JobLinkedList::~JobLinkedList() // version 3
{
  for(int i = 0; i < this->count; i++)
    this->remove(0);
}
// is this OK?
```

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 **`nullptr`** pointer.
- Check whether you need a destructor (or a copy constructor or an assignment operator) when your class has a pointer member.

Outline

- Basic ideas
- Lists: **`class JobList`**
- Linked lists: **`JobLinkedList`**
- **More data structures**

Stacks and queues

- A **stack** is a special list. A **queue** is another special list.
- Nodes cannot be inserted/removed at any place we want.
  - Stack: last-in-first-out (**LIFO**). A node can be inserted and removed only at the **top** of the stack.
  - Queue: first-in-first-out (**FIFO**). A node can be inserted only at the **tail** and removed only at the **head**.

Stacks and queues

- Many real-life situations can be modeled as stacks.
  - The poker game solitaire;
  - The Hanoi tower;
  - Function calls in your programs;
  - Calculators;
  - Graph traversal: depth-first search (DFS).
- Many real-life situations can be modeled as 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.
  - 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
// protected: we want to hide insert()
// and remove() inherited from JobLinkedList
{
public:
  JobStack();
  ~JobStack();
  bool push(Job job);
  bool pop();
  void print();
};
```

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

JobStack::~JobStack()
{
}
```

JobStack

```
void JobStack::print() // You need print() due to protected inheritance
{
  JobLinkedList::print();
}

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

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

A main function

```
int main()
{
  JobStack s;
  Job j1("Programming", 10);
  Job j2("Accounting", 20);
  s.push(j1);
  s.push(j2);
  s.print();
  s.pop();
  s.print();

  return 0;
}
```

Remarks

- The class **JobStack** is indeed a stack. It is **safe and effective**.
- However, it is **not very efficient**.
  - Operations are executed through another class.
  - **push()** and **pop()** are both $O(n)$.
- To remedy this, we may add a new member to **JobStack**:
  - With **Job* tail** (as a new instance variable), **push()** and **pop()** can be both $O(1)$.
- Be careful that **insert()** and **remove()** should be **hided**.
  - If you use public inheritance, you may override them as private member functions.
- 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;
  // …
}
```