Implementation of algorithms and data structures

8. seminar

Jirka Fink

https://ktiml.mff.cuni.cz/~fink/

Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics
Charles University in Prague

Summer semestr 2021/22
Last change 13. dubna 2022

Linked lists

Termination

- Last element points to NULL/nullptr/None (Null-terminated)
- Last element points to the first one (Circular)

Linked lists

Termination

- Last element points to NULL/nullptr/None (Null-terminated)
- Last element points to the first one (Circular)

How to store next/previous pointers

- Pointers are stored inside object (Intrusive)

```
class MyClass {
   string name; # Variables we need in the class
   node *next, *prev;
};
```

Termination

- Last element points to NULL/nullptr/None (Null-terminated)
- Last element points to the first one (Circular)

How to store next/previous pointers

- Pointers are stored inside object (Intrusive)

```
class MyClass {
   string name; # Variables we need in the class
   node *next, *prev;
};
```

- Pointers are stored outside object (Non-intrusive)

```
class MyClass {
   string name;
};
list<MyClass> my_list;
```

Termination

- Last element points to NULL/nullptr/None (Null-terminated)
- Last element points to the first one (Circular)

How to store next/previous pointers

- Pointers are stored inside object (Intrusive)

```
class MyClass {
   string name; # Variables we need in the class
   node *next, *prev;
};
```

- Pointers are stored outside object (Non-intrusive)

```
class MyClass {
   string name;
};
list<MyClass> my_list;
```

Goal

We can easily add pointers into a class, but we do not want to repetitively write functions working with pointers.
How to design a general API?

Linked list: NULL-terminated lists

```
struct node {
   node *next, *prev;
};
struct list {
   node *first, *last;
};
void list_add_tail(list *l, node *n) {
   n->next = NULL;
   n->prev = l->last;
   if(l->last)
      l->last->next = n;
   else
      l->first = n;
   l->last = n;
}
void list_remove(list *l, node *n) {
   if(n->prev)
      n->prev->next = n->next;
   else
      l->first = n->next;
   if(n->next)
      n->next->prev = n->prev;
   else
      l->last = n->prev;
}
```

Linked list: Circular lists

```
struct node {
    node *next, *prev;
};
struct list {
    node head;
};

void list_init(list *l) {
    l->head.next = l->head.prev = &l->head;
}

void list_insert_after(node *what, node *after) {
    node *before = after->next;
    what->next = before;
    what->prev = after;
    before->prev = what;
    after->next = what;
}

void list_remove(node *n) {
    node *before = n->prev;
    node *after = n->next;
    before->next = after;
    after->prev = before;
}
```

Non-intrusive Null-terminated list: C++ using templates

```
template<class T>
class list {
   struct node {
      T data;
      node *next, *prev;
   };
   node *first, *last;

public:
   using iterator = node*;
   iterator insert(iterator pos, const T& value);
   iterator erase(iterator pos);
   // ...
};

std::list<std::string> fruits {"orange", "apple", "raspberry"};

// To store one instance in multiple lists, create lists of pointers
std::list<MyClass*> list1, list2;
```

Intrusive list without templates

```
struct node { node *next, *prev; };
struct list { node head; };

node* list_get_first(list *l) {
   return l->head.next;
}

struct company {
   char *name;
   list employees, consumers;
};

struct employee {
   node n; // Assume that this node can be the first member variable
   char *name;
}

void company_add_employee(company *c, employee *e) {
   list_add_tail(&c->employees, &e->n);
}

employee* company_get_first_employee(company *c) {
   return (employee*) list_get_first(&c->employees);
}
```

Intrusive list without templates

```
#define offsetof(TYPE, MEMBER) ((size_t)&((TYPE *)0)->MEMBER)

#define container_of(PTR, TYPE, MEMBER) \
   ((TYPE *)((char *)PTR - offsetof(TYPE, MEMBER)))

struct company {
   char *name;
   list employees, consumers;
};

struct consumer {
   char *name;
   node n; // Assume that this node cannot be the first member variable
};

consumer* consumer_from_node(node *node_in_consumer) {
   return container_of(node_in_consumer, consumer, n);
}

consumer* company_get_first_consumer(company *c) {
   return consumer_from_node(list_get_first(&c->consumers));
}
```

Intrusive NULL-terminated list without memory magic

```
template <class T>
struct node {
   T *next, *prev;
};

template <class T, node<T> T::*M> // Member pointer
class list {
   T *first = nullptr, *last = nullptr;

public:
   T* begin() {
   return first;
   }

   void add_head(T *n) {
   if(first)
      (first->*M).prev = n;
   (n->*M).next = first;
   first = n;
   }
}
```

Intrusive circular list using templates

```
struct node {
   node *next = nullptr, *prev = nullptr;
};

template <class T, node T::*M>
class list {
   node head;

   T* container_of(node *n) {
   const size_t offset = (size_t)(&(((T*)nullptr)->*M));
   return (T*)((char *)n - offset);
   }

public:
   T* begin() {
   return head.next != &head ? container_of(head.next) : nullptr;
   }
}
```

Using intrusive lists: Sparse matrix

```
class matrix_element {
   node row, column;
};

class matrix_row {
   node n;
   list<matrix_element, &matrix_element::row> row;
};

class matrix_column {
   node n;
   list<matrix_element, &matrix_element::column> column;
};

class matrix {
   list<matrix_row, &matrix_row::n> rows;
   list<matrix_column, &matrix_column::n> columns;
};
```

Using intrusive lists: Tree

Example in C

```c
struct tree_node{
   list children;
   node siblings; // Node for the list of children
   struct tree_node *parent;
};

struct tree {
   tree_node head;
};
```

Example in C++

```cpp
class tree {
   struct tree_node {
   tree_node *parent;
   node<tree_node> siblings;
   list<tree_node, &tree_node::siblings> children;
   };
   tree_node head;
};
```

Circular vs. NULL-terminated

- If it is possible to use a circular linked list, it is better than NULL-terminated

Summary

Circular vs. NULL-terminated

- If it is possible to use a circular linked list, it is better than NULL-terminated

Non-intrusive `list<MyClass*>`

- Element can be in multiple lists
- Iterator cannot be obtained from a pointer to MyClass (e.g. to remove it from a list)
- Requires more memory and worser cache-efficiency

Summary

Circular vs. NULL-terminated

- If it is possible to use a circular linked list, it is better than NULL-terminated

Non-intrusive `list<MyClass*>`

- Element can be in multiple lists
- Iterator cannot be obtained from a pointer to MyClass (e.g. to remove it from a list)
- Requires more memory and worser cache-efficiency

Non-intrusive `list<MyClass>`

- Element is in one list only
- Some languages does not support obtaining an iterator from an instance
- Unsupported by some languages

Summary

Circular vs. NULL-terminated

- If it is possible to use a circular linked list, it is better than NULL-terminated

Non-intrusive `list<MyClass*>`

- Element can be in multiple lists
- Iterator cannot be obtained from a pointer to MyClass (e.g. to remove it from a list)
- Requires more memory and worser cache-efficiency

Non-intrusive `list<MyClass>`

- Element is in one list only
- Some languages does not support obtaining an iterator from an instance
- Unsupported by some languages

Intrusive

- Element can be in as many lists as it contains next pointers
- May require memory magic which is not available in some languages
- Appropriate for advanced algorithms which encapsulate lists
- Inappropriate between modules of complex programs

Do not use linked lists for everything

Array

- Significantly smaller overhead
- Better usage of cache, especially for small elements
- Sequential access does not require conditional jumps

Do not use linked lists for everything

Array

- Significantly smaller overhead
- Better usage of cache, especially for small elements
- Sequential access does not require conditional jumps

Hashing

- Expected complexity $O(1)$ for operations find, insert and delete
- Requires a good hashing function

Do not use linked lists for everything

Array

- Significantly smaller overhead
- Better usage of cache, especially for small elements
- Sequential access does not require conditional jumps

Hashing

- Expected complexity $O(1)$ for operations find, insert and delete
- Requires a good hashing function

Search trees

- Worst-case complexity $O(\log n)$
- Requires fast comparison of elements
- Support interval queries, finding successor element, etc.

Do not use linked lists for everything

Array

- Significantly smaller overhead
- Better usage of cache, especially for small elements
- Sequential access does not require conditional jumps

Hashing

- Expected complexity $O(1)$ for operations find, insert and delete
- Requires a good hashing function

Search trees

- Worst-case complexity $O(\log n)$
- Requires fast comparison of elements
- Support interval queries, finding successor element, etc.

Many other data structures

- Heaps
- Union-find
- ...

Exception handling

Popis

Exception handling is the process of responding to the occurrence of exceptions – anomalous or exceptional conditions requiring special processing – during the execution of a program.

Exception handling

Popis

Exception handling is the process of responding to the occurrence of exceptions – anomalous or exceptional conditions requiring special processing – during the execution of a program.

Examples of exceptions

- Index out of range of an array
- Retrieve the first element of an empty list
- Searched element does not exist
- Square root of a negative number
- Determine a circle from three colinear points
- File access without sufficient permission
- Error in reading data
- Invalid user input
- Insufficient amount of memory

Methods of handling exceptions

- Ignore, leads to an undefined behavior
  `vector::operator[]`, `list::pop_front`
- Throw an exception, call longjmp
  `vector::at`
- Call a function
  `set_terminate`, signal handling
- Set a status variable, e.g. `errno`
  `fopen`, `scanf`
- Return an invalid value, e.g. NULL, a dummy object
  `find`
- Return a valid value
  `double sqrt(double x){ if(x < 0)return 0; ... }`
- Return a pair of a status and a result
  `pair<bool, int> find(x){ return exist(x)? make_pair(true, search(x)): make_pair(false, 0); }`
- Use goto to an error label
- Terminate the program, e.g. using functions `abort` or `assert`

Exception handling: Improper and rare solutions

Return a valid value

- A calling function cannot recognize an error
- The error is propagated which makes it harder to find

Exception handling: Improper and rare solutions

Return a valid value

- A calling function cannot recognize an error
- The error is propagated which makes it harder to find

Set a status variable

- Hard to manage
- Requires detailed documentation
- Data race conditions

Exception handling: Improper and rare solutions

Return a valid value

- A calling function cannot recognize an error
- The error is propagated which makes it harder to find

Set a status variable

- Hard to manage
- Requires detailed documentation
- Data race conditions

Use goto to an error label

- Can be used only inside a function
- In rare situations, it may simplify a complex if-else statements

Exception handling: Improper and rare solutions

Return a valid value

- A calling function cannot recognize an error
- The error is propagated which makes it harder to find

Set a status variable

- Hard to manage
- Requires detailed documentation
- Data race conditions

Use goto to an error label

- Can be used only inside a function
- In rare situations, it may simplify a complex if-else statements

Call a function

Only for asynchronous or external events, e.g. signal handling.

Exception handling: Invalid arguments in public methods (bugs)

Throw an exception

- Safe
- Checking conditions delays computation
- Requires catching exceptions
- Requires documentation of used exceptions

Exception handling: Invalid arguments in public methods (bugs)

Throw an exception

- Safe
- Checking conditions delays computation
- Requires catching exceptions
- Requires documentation of used exceptions

Assert in a debug mode, ignore in a release mode

- Fast (in release mode)
- May lead to an undefined behavior
- Requires documentation of conditions on arguments

Exception handling: Unavoidable situations

Return an invalid value

Prefer when

- the calling function is expected to handle
- the exception is a natural output (e.g. no element satisfies a given condition)

If an invalid values does not exist, return a pair with a status.

Exception handling: Unavoidable situations

Return an invalid value

Prefer when

- the calling function is expected to handle
- the exception is a natural output (e.g. no element satisfies a given condition)

If an invalid values does not exist, return a pair with a status.

Throw an exception

Prefer when

- the exception may jump through many functions
- the situation is very rare
- many different types of errors may occur

Exception handling: Unavoidable situations

Assert in a debug mode, ignore in a release mode

- Fast
- Easy to debug
- Cause undefined behavior in release mode

Exception handling: Unavoidable situations

Assert in a debug mode, ignore in a release mode

- Fast
- Easy to debug
- Cause undefined behavior in release mode

A general rule

- Write documentation
- Always follow a project policy!

Exception handling: Exception safety

Example of an issue

Consider a function `push_back` inserting into a `vector<MyClass>` when its array is full. Reallocation moves all element into a new memory but move constructor throws an exception. What the function `push_back` should do?

Example of an issue

Consider a function `push_back` inserting into a `vector<MyClass>` when its array is full. Reallocation moves all element into a new memory but move constructor throws an exception. What the function `push_back` should do?

Exception safety

- No-throw guarantee: Operations are guaranteed to succeed and satisfy all requirements even in exceptional situations. If an exception occurs, it will be handled internally and not observed by clients.
- Strong exception safety: Operations can fail, but failed operations are guaranteed to have no side effects, so all data retains their original values.
- Basic exception safety: Partial execution of failed operations can cause side effects, but all invariants are preserved and there are no resource leaks (including memory leaks). Any stored data will contain valid values, even if they differ from what they were before the exception.
- No exception safety: No guarantees are made.