Khoa CNTT – ĐH Khoa học Tự nhiên
TP HCM, tháng 02/2022

Kỹ thuật lập trình

Các cấu trúc dữ liệu cơ bản

Nguyễn Trọng Việt

Mục tiêu

Sau bài học, sinh viên có thể:

- Cài đặt và sử dụng danh sách liên kết
- Cài đặt và sử dụng ngăn xếp
- Cài đặt và sử dụng hàng đợi
- Liệt kê các thao tác cơ bản trên cây nhị phân tìm kiếm

Nội dung

- Danh sách liên kết
- Ngăn xếp
- Hàng đợi
- Cây nhị phân tìm kiếm
- Các cấu trúc dữ liệu khác

Nội dung

- Danh sách liên kết
- Ngăn xếp
- Hàng đợi
- Cây nhị phân tìm kiếm
- Các cấu trúc dữ liệu khác

Danh sách liên kết là gì ?

- Là một cấu trúc dữ liệu dạng container chứa các phần tử (còn gọi là nút) được bố trí tuần tự, không liên tiếp
- Trong đó:
  - Các nút liên kết với nhau bằng con trỏ
  - Head: con trỏ đến nút đầu danh sách
  - Tail: con trỏ đến nút cuối danh sách
- Phân loại:
  - Danh sách liên kết đơn
  - Danh sách liên kết đôi
  - Danh sách liên kết vòng
  - …

Danh sách liên kết

Danh sách liên kết đơn

Danh sách liên kết đôi

Danh sách liên kết vòng

Danh sách liên kết

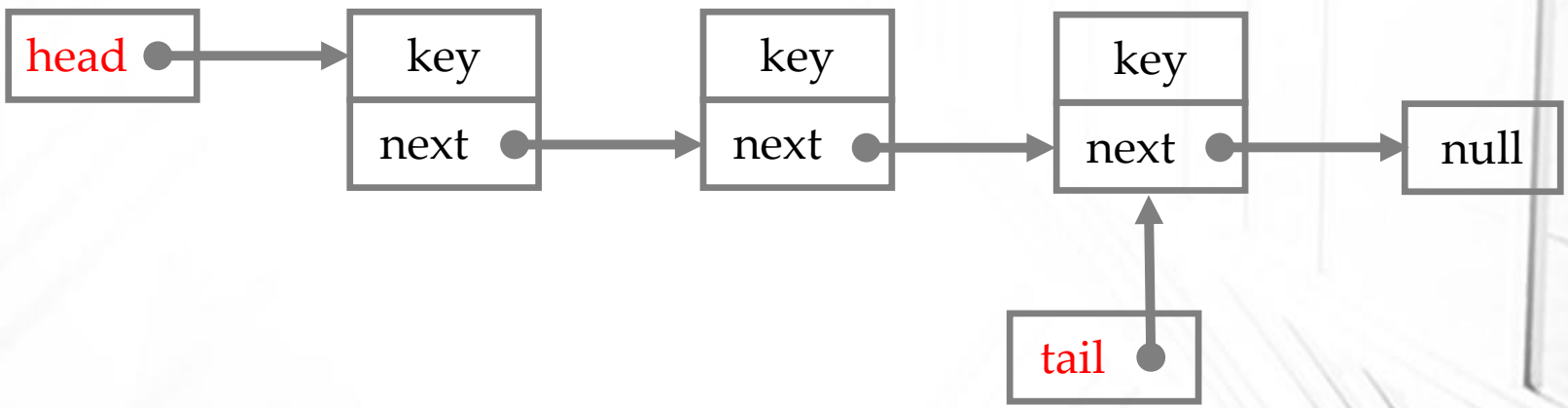
Danh sách liên kết đơn

Danh sách liên kết đôi

Danh sách liên kết vòng

Danh sách liên kết đơn

- Là danh sách liên kết trong đó nút trước liên kết với nút kế tiếp bằng con trỏ
- Mỗi nút gồm 2 trường:
  - Khóa (key) chứa giá trị của nút
  - Con trỏ đến nút kế tiếp



Danh sách liên kết đơn

- Nút:

```
struct SNode
{
    int key;
    SNode* pNext;
};
```

- Danh sách:

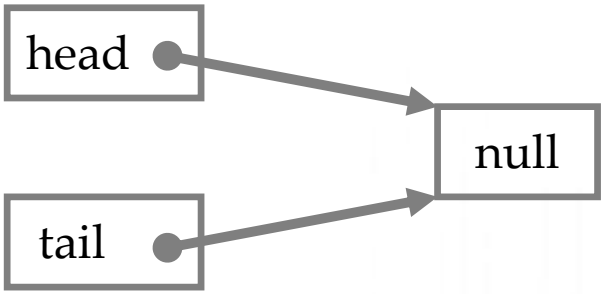
```
struct SList
{
    SNode* pHead;
    SNode* pTail;
};
```

Danh sách liên kết đơn

- Các thao tác:
  - Khởi tạo
  - Kiểm tra danh sách rỗng
  - Tạo nút mới
  - Thêm phần tử
  - Xóa phần tử
  - Duyệt danh sách
  - Xóa danh sách
  - Thao tác khác

Danh sách liên kết đơn

- Khởi tạo:

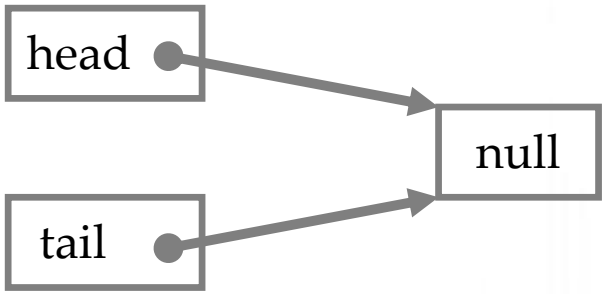



```
void initialize(SList** w_list)
{
    *w_list = new SList;
    (*w_list)->pHead = nullptr;
    (*w_list)->pTail = nullptr;
}
```

```
void initialize(SList** w_list)
{
    *w_list = new SList{nullptr, nullptr};
}
```

Danh sách liên kết đơn

- Kiểm tra danh sách rỗng:




```
bool isEmpty(const SList* r_list)
{
    ...
}
```

Danh sách liên kết đơn

- Tạo nút mới

```
SNode* newNode(int value)
{
    SNode* node = new SNode;
    node->key = value;
    node->pNext = nullptr;

    return node;
}
```

```
SNode* newNode(int value)
{
    return new SNode{ value, nullptr };
}
```

Danh sách liên kết đơn

- Thêm phần tử:
  - Thêm vào đầu danh sách
  - Thêm vào cuối danh sách
  - *Thêm vào sau 1 nút*

```
SNode* pushFront(SList* u_list, int r_val);

SNode* pushBack(SList* u_list, int r_val);

SNode* insertAfter(SNode* u_node, int r_val);
```

Danh sách liên kết đơn

- Xóa phần tử:
  - Xóa đầu danh sách
  - Xóa cuối danh sách
  - *Xóa sau 1 nút*

```
void popFront(SList* u_list);

void popBack(SList* u_list);

void removeAfter(SNode* u_node);
```

Danh sách liên kết đơn

- Duyệt danh sách

```
void print(SList* r_list)
{
    SNode* node = r_list->pHead;
    while( node ) {
        printf(" - val: %d\n", node->key);
        node = node->pNext;
    }
}
```

```
void print(SList* r_list)
{
    for( SNode* node = r_list->pHead ;
        node ;
        node = node->pNext )
    {
        printf(" - val: %d\n", node->key);
    }
}
```

Danh sách liên kết đơn

- Tìm phần tử

```
SNode* search(SList* r_list, int r_val)
{
    ...
}
```

Danh sách liên kết đơn

- Thêm nút mới sau 1 nút
- Xóa nút sau 1 nút khác

```
SNode* insertAfter(SNode* u_node, int r_val)
{
    ...
}

void removeAfter(SNode* u_node)
{
    ...
}
```

Danh sách liên kết đơn

- Xóa danh sách

```
void deinitialize(SList** w_list)
{
    ...
}
```

Danh sách liên kết đơn

- Giải trí
  - Khởi tạo danh sách rỗng
  - Lần lượt thêm 4, 6, 8 vào đầu danh sách
  - Lần lượt thêm 3, 5, 7, 9 vào cuối danh sách
  - Thêm 2 sau 5 và thêm 1 trước 4
  - Xóa phần tử trước 7
  - Xóa phần tử trước nút cuối cùng
  - Sau mỗi thao tác in danh sách ra màn hình

Danh sách liên kết đơn

- Các thao tác khác
  - Trộn 2 danh sách
  - Hoán đổi 2 danh sách
  - Nghịch đảo danh sách
  - *Sắp xếp danh sách*

```
void merge(SList* u_list_a, SList* u_list_b);

void swap(SList* u_list_a, SList* u_list_b);

void reverse(SList* u_list);

void sort(SList* u_list);
```

Danh sách liên kết

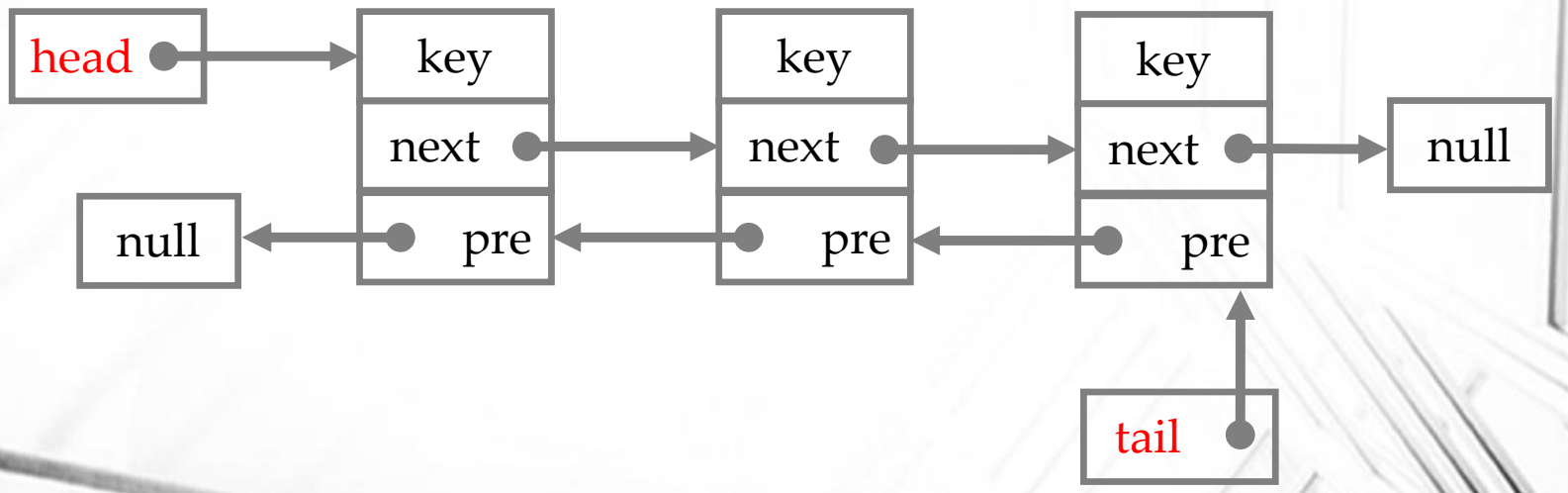
Danh sách liên kết đơn

Danh sách liên kết đôi

Danh sách liên kết vòng

Danh sách liên kết đôi

- Là danh sách liên kết trong đó mỗi nút liên kết với nút liền trước và liền sau bằng con trỏ
- Mỗi nút gồm 3 trường:
  - Khóa (key) chứa giá trị của nút
  - Con trỏ đến nút liền sau
  - Con trỏ đến nút liền trước



Danh sách liên kết đôi

- Nút:

```
struct DNode
{
    int key;
    DNode* pNext;
    DNode* pPrevious;
};
```

- Danh sách:

```
struct DList
{
    DNode* pHead;
    DNode* pTail;
};
```

Danh sách liên kết đôi

- Các thao tác:
  - Khởi tạo
  - Kiểm tra danh sách rỗng
  - Tạo nút mới
  - Thêm phần tử
  - Xóa phần tử
  - Duyệt danh sách
  - Xóa danh sách
  - Thao tác khác

Danh sách liên kết

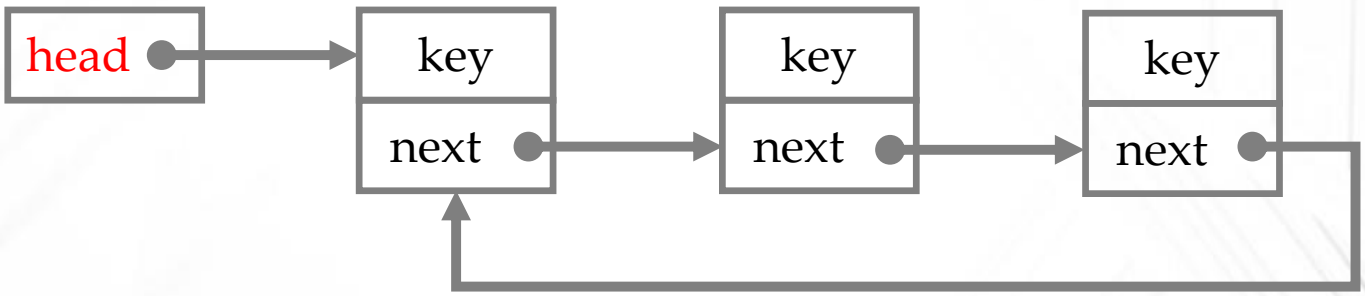
Danh sách liên kết đơn

Danh sách liên kết đôi

Danh sách liên kết vòng

Danh sách liên kết vòng

- Là danh sách liên kết trong đó mỗi nút liên kết với nút liền sau nó, riêng nút cuối liên kết với nút đầu tiên
- Mỗi nút gồm 2 trường:
  - Khóa (key) chứa giá trị của nút
  - Con trỏ đến nút kế tiếp



Danh sách liên kết vòng

- Nút:

```
struct CNode
{
    int key;
    CNode* pNext;
};
```

- Danh sách:

```
struct CList
{
    CNode* pHead;
};
```

Danh sách liên kết vòng

- Các thao tác:
  - Khởi tạo
  - Kiểm tra danh sách rỗng
  - Tạo nút mới
  - Thêm phần tử
  - Xóa phần tử
  - Duyệt danh sách
  - Xóa danh sách
  - Thao tác khác

Thư viện chuẩn

Thư viện STL

`std::forward_list`

- Là một kiểu container của thư viện STL
- Lớp template cài đặt **danh sách liên kết đơn**
- Hỗ trợ insert và remove nhanh chóng bất kì phần tử nào trên danh sách
- Không hỗ trợ truy xuất phần tử ngẫu nhiên

Thư viện STL

`std::forward_list`

Phương thức	Chức năng
`empty`	Kiểm tra danh sách rỗng
`clear`	Xóa nội dung danh sách
`insert_after`	Chèn phần tử mới sau phần tử
`erase_after`	Xóa phần tử sau phần tử
`push_front`	Chèn phần tử mới đầu danh sách
`pop_front`	Xóa phần tử đầu danh sách
`merge`	Trộn 2 danh sách
`swap`	Hoán đổi nội dung 2 danh sách
`sort`	Sắp xếp nội dung danh sách
...	...

Thư viện STL

`std::list`

- Là một kiểu container của thư viện STL
- Lớp template cài đặt **danh sách liên kết đôi**
- Hỗ trợ insert và remove nhanh chóng bất kì phần tử nào trên danh sách
- Không hỗ trợ truy xuất phần tử ngẫu nhiên

Thư viện STL

`std::list`

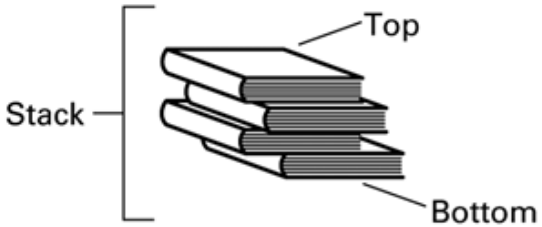
Phương thức	Chức năng
`empty`	Kiểm tra danh sách rỗng
`clear`	Xóa nội dung danh sách
`insert_after`	Chèn phần tử mới sau phần tử
`erase_after`	Xóa phần tử sau phần tử
`push_front`	Chèn phần tử mới đầu danh sách
`pop_front`	Xóa phần tử đầu danh sách
`merge`	Trộn 2 danh sách
`swap`	Hoán đổi nội dung 2 danh sách
`sort`	Sắp xếp nội dung danh sách
...	...

Nội dung

- Danh sách liên kết
- Ngăn xếp
- Hàng đợi
- Cây nhị phân tìm kiếm
- Các cấu trúc dữ liệu khác

Ngăn xếp là gì ?

- Là một cấu trúc dữ liệu dạng container hỗ trợ 2 thao tác chính:
  - Push: đẩy 1 phần tử mới vào đỉnh ngăn xếp
  - Pop: lấy ra 1 phần tử từ đỉnh ngăn xếp
- Nguyên tắc: LIFO – Last In First Out
- Cài đặt:
  - Sử dụng mảng
  - Sử dụng danh sách liên kết đơn



Ngăn xếp

Thao tác:

- Khởi tạo ngăn xếp
- Kiểm tra ngăn xếp rỗng/đầy
- Đẩy phần tử mới vào đỉnh ngăn xếp
- Lấy phần tử từ đỉnh ngăn xếp
- Đọc nội dung đỉnh ngăn xếp
- Hủy ngăn xếp

Ngăn xếp

Sử dụng mảng

Sử dụng danh sách liên kết

Ứng dụng

Ngăn xếp

Sử dụng mảng

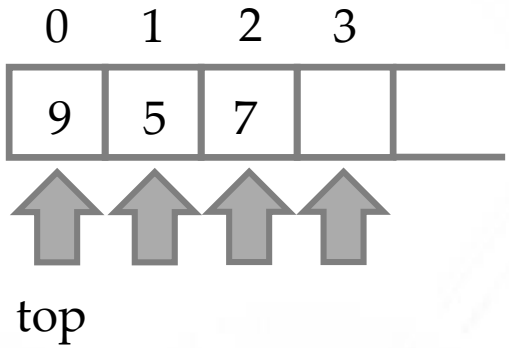
Sử dụng danh sách liên kết

Ứng dụng

Cài đặt ngăn xếp bằng mảng

Ý tưởng:

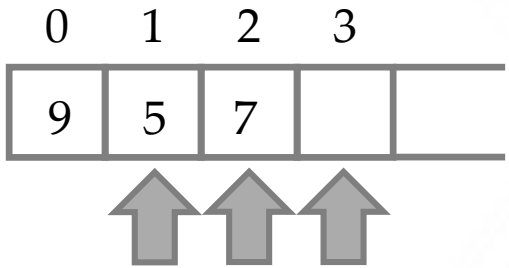
- Sử dụng mảng 1 chiều để lưu trữ các phần tử
- Sử dụng biến chỉ mục `top` để đánh dấu đỉnh ngăn xếp
- Push: thêm phần tử tại `top` và tăng `top`
- Pop: xóa phần tử tại `top` và giảm `top`



Cài đặt ngăn xếp bằng mảng

Ý tưởng:

- Sử dụng mảng 1 chiều để lưu trữ các phần tử
- Sử dụng biến chỉ mục `top` để đánh dấu đỉnh ngăn xếp
- Push: thêm phần tử tại `top` và tăng `top`
- Pop: xóa phần tử tại `top` và giảm `top`



Cài đặt ngăn xếp bằng mảng

Khai báo ngăn xếp:

```
#define MAX_SIZE 100

struct Stack
{
    int data[MAX_SIZE];
    int top;
};
```

Cài đặt ngăn xếp bằng mảng

- Khởi tạo ngăn xếp

```
void initialize(Stack** stack)
{
    *stack = new Stack;
    (*stack)->top = 0;
}
```

- Hủy ngăn xếp

```
void deinitialize(Stack** stack)
{
    ...
}
```

Cài đặt ngăn xếp bằng mảng

- Kiểm tra ngăn xếp rỗng/đầy

```
bool isEmpty(const Stack* stack)
{
    ...
}

bool isFull(const Stack* stack)
{
    ...
}
```

Cài đặt ngăn xếp bằng mảng

- Đẩy phần tử mới vào đỉnh ngăn xếp
- Lấy phần tử từ đỉnh ngăn xếp
- Đọc nội dung đỉnh ngăn xếp

```
void push(Stack* stack, int val)
{
    ...
}

void pop(Stack* stack)
{
    ...
}

int top(const Stack* stack)
{
    ...
}
```

Cài đặt ngăn xếp bằng mảng

Ưu điểm:

- Đơn giản
- Cài đặt nhanh chóng

Nhược điểm:

- Push vào 1 ngăn xếp đã đầy !!!

Ngăn xếp

Sử dụng mảng

Sử dụng danh sách liên kết

Ứng dụng

Cài đặt ngăn xếp bằng danh sách liên kết

Ý tưởng:

- Sử dụng danh sách liên kết đơn để lưu trữ các phần tử
- Sử dụng con trỏ `top = head` để trỏ đến phần tử trên đỉnh ngăn xếp
- Push: thêm phần tử đầu danh sách
- Pop: xóa phần tử đầu danh sách

Cài đặt ngăn xếp bằng danh sách liên kết

Khai báo ngăn xếp:

```
struct SNode
{
    int key;
    SNode* pNext;
};

struct Stack
{
    SNode* top;
};
```

Cài đặt ngăn xếp bằng danh sách liên kết

- Khởi tạo ngăn xếp
- Kiểm tra ngăn xếp rỗng
- Đẩy phần tử mới vào đỉnh ngăn xếp
- Xóa phần tử từ đỉnh ngăn xếp
- Đọc nội dung đỉnh ngăn xếp
- Hủy ngăn xếp

```
void initialize(Stack** stack);

bool isEmpty(const Stack* stack);

void push(Stack* stack, int val);

void pop(Stack* stack);

int top(const Stack* stack);

void deinitialize(Stack** stack);
```

Cài đặt ngăn xếp bằng danh sách liên kết

Ưu điểm:

- Không bị giới hạn số lượng phần tử

Nhược điểm:

- Cài đặt phức tạp

Ngăn xếp

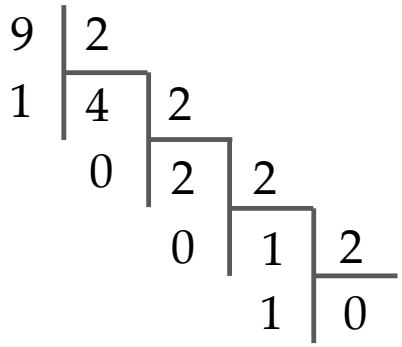
Sử dụng mảng

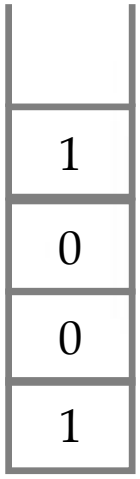
Sử dụng danh sách liên kết

Ứng dụng

Ứng dụng ngăn xếp

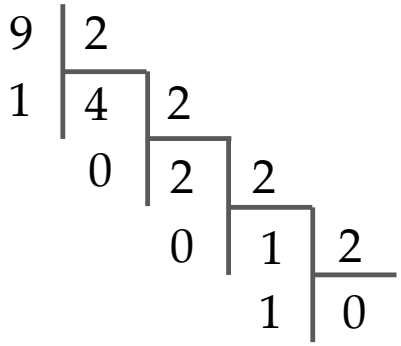
Chuyển hệ cơ số

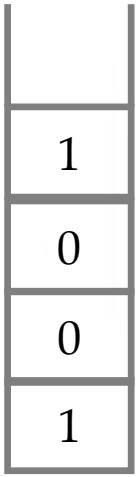


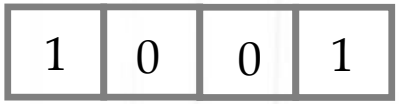


Ứng dụng ngăn xếp

Chuyển hệ cơ số







Ứng dụng ngăn xếp

Tính giá trị biểu thức hậu tố (suffix)

- Biểu thức trung tố (infix)

```
4 + 6 * 9

(4 + 6) * 9
```

- Biểu thức hậu tố (suffix)

```
4 6 9 * +

4 6 + 9 *
```

Ứng dụng ngăn xếp

Tính giá trị biểu thức hậu tố (suffix)

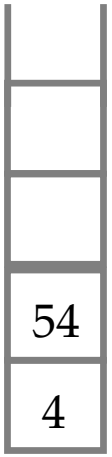
4 6 9 * +

9 * 6 = 54

Ứng dụng ngăn xếp

Tính giá trị biểu thức hậu tố (suffix)

4	6	9	*	+

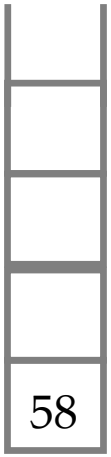


54 + 4 = 58

Ứng dụng ngăn xếp

Tính giá trị biểu thức hậu tố (suffix)

4	6	9	*	+



Thư viện chuẩn

Thư viện STL

- `std::stack`

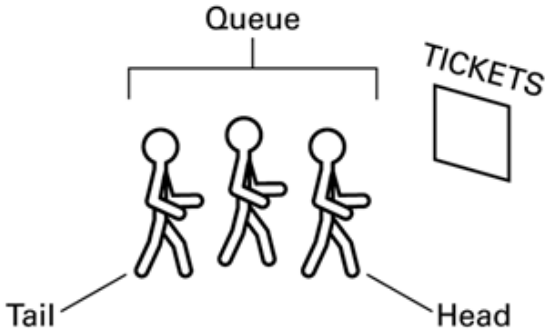
Phương thức	Chức năng
`empty`	Kiểm tra danh sách rỗng
`top`	Truy xuất phần tử trên đỉnh ngăn xếp
`push`	Đẩy phần tử mới vào đỉnh ngăn xếp
`pop`	Xóa phần tử trên đỉnh ngăn xếp
...	...

Nội dung

- Danh sách liên kết
- Ngăn xếp
- Hàng đợi
- Cây nhị phân tìm kiếm
- Các cấu trúc dữ liệu khác

Hàng đợi là gì ?

- Là một cấu trúc dữ liệu dạng container hỗ trợ 2 thao tác chính:
  - Enqueue: thêm phần tử mới vào cuối hàng đợi
  - Dequeue: lấy 1 phần tử từ đầu hàng đợi
- Nguyên tắc: FIFO – First In First Out



Hàng đợi

Các thao tác:

- Khởi tạo hàng đợi
- Kiểm tra hàng đợi rỗng/đầy
- Thêm phần tử vào cuối hàng đợi
- Lấy phần tử từ đầu hàng đợi
- Đọc nội dung phần tử đầu hàng đợi
- Hủy hàng đợi

Hàng đợi

- Phân loại:
  - Hàng đợi truyền thống
  - Hàng đợi 2 đầu
  - Hàng đợi có độ ưu tiên
- Cài đặt:
  - Sử dụng mảng
  - Sử dụng danh sách liên kết đơn

Hàng đợi

Hàng đợi truyền thống

Hàng đợi 2 đầu

Hàng đợi có độ ưu tiên

Hàng đợi

Hàng đợi truyền thống

Hàng đợi 2 đầu

Hàng đợi có độ ưu tiên

Hàng đợi truyền thống

Cài đặt:

- Sử dụng mảng
- Sử dụng danh sách liên kết đơn

Thao tác:

- Khởi tạo hàng đợi
- Kiểm tra hàng đợi rỗng/đầy
- Thêm phần tử mới vào cuối hàng đợi
- Lấy phần tử từ đầu hàng đợi
- Đọc nội dung phần tử đầu hàng đợi
- Hủy hàng đợi

Hàng đợi truyền thống

```
// use array
#define MAX_SIZE 100

struct Queue
{
    int data[MAX_SIZE];
    int head;
    int tail;
};
```

```
// use singly linked list
struct SNode
{
    int key;
    SNode* pNext;
};

struct Queue
{
    SNode* head;
    SNode* tail;
};
```

```
void initialize(Queue** queue);

bool isEmpty(const Queue* queue);

void push(Queue* queue, int val);

void pop(Queue* queue);

int front(const Queue* queue);

void deinitialize(Queue** queue);
```

Hàng đợi

Hàng đợi truyền thống

Hàng đợi 2 đầu

Hàng đợi có độ ưu tiên

Hàng đợi 2 đầu

- Deque
- Là mở rộng của hàng đợi truyền thống, cho phép thêm và xóa phần tử ở cả 2 đầu

Thao tác:

- Khởi tạo hàng đợi
- Kiểm tra hàng đợi rỗng/đầy
- Thêm phần tử mới vào cuối/đầu hàng đợi
- Lấy phần tử từ đầu/cuối hàng đợi
- Đọc nội dung phần tử đầu/cuối hàng đợi
- Hủy hàng đợi

Hàng đợi 2 đầu

```
void initialize(Deque** deque);

bool isEmpty(const Deque* deque);

void pushFront(Deque* deque, int val);

void pushBack(Deque* deque, int val);

void popFront(Deque* deque);

void popBack(Deque* deque);

int front(const Deque* deque);

int back(const Deque* deque);

void deinitialize(Deque** deque);
```

Hàng đợi

Hàng đợi truyền thống

Hàng đợi 2 đầu

Hàng đợi có độ ưu tiên

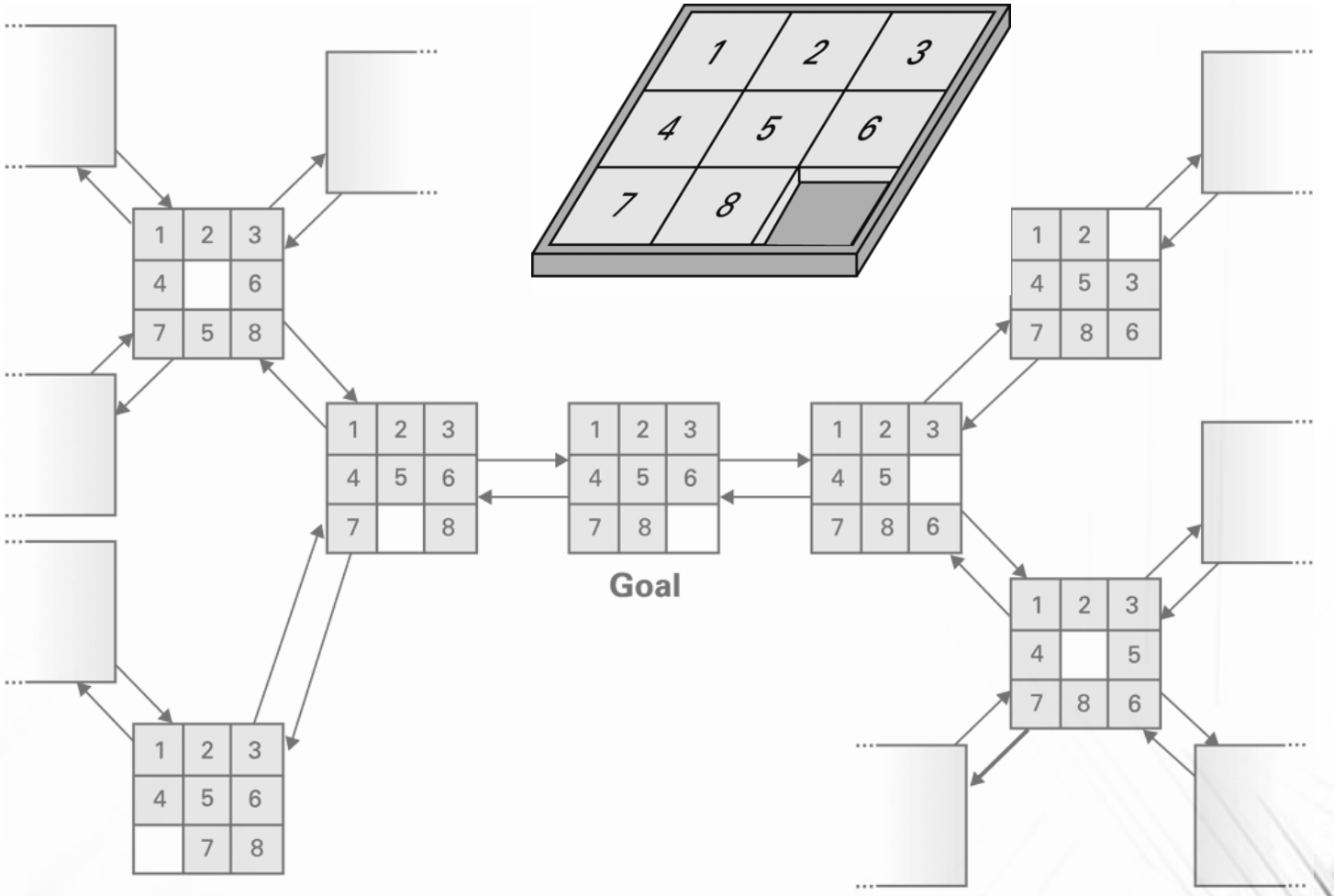
Hàng đợi có độ ưu tiên

- Priority queue
- Là mở rộng của hàng đợi truyền thống, trong đó:
  - Mỗi phần tử được đính kèm độ ưu tiên
  - Phần tử có độ ưu tiên cao nhất sẽ được ưu tiến lấy ra khỏi hàng đợi trước
  - Các phần tử có cùng độ ưu tiên được lấy ra theo thứ tự FIFO
- Thao tác:
  - Khởi tạo hàng đợi
  - Kiểm tra hàng đợi rỗng/đầy
  - Thêm phần tử vào hàng đợi
  - Lấy ra phần tử có độ ưu tiên cao nhất
  - Hủy hàng đợi

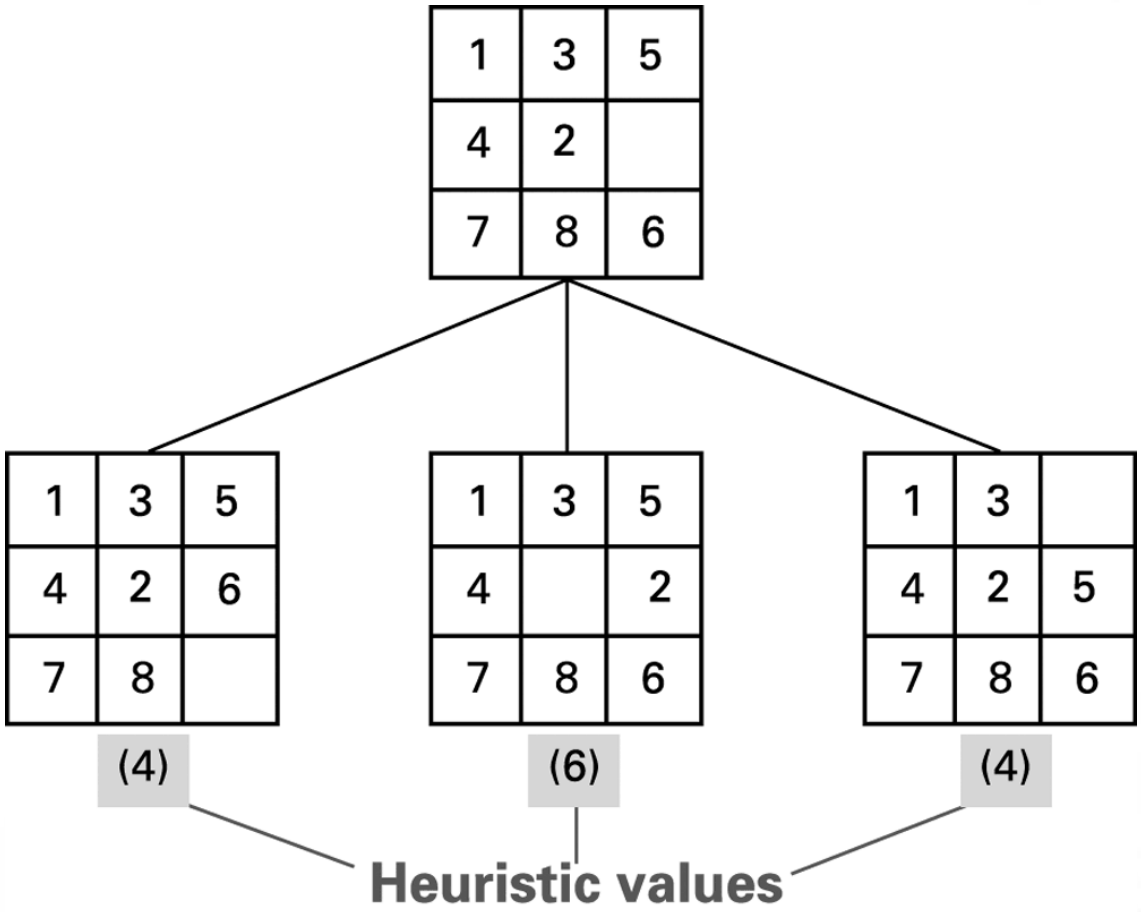
Ứng dụng

- Thuật toán Dijkstra trong tìm kiếm đường đi ngắn nhất
- Thuật toán Prime trong xác định cây khung tối thiểu
- Mã hóa Huffman
- Thuật toán tìm kiếm ưu tiên lựa chọn tốt nhất (Best-First search) trong Trí tuệ nhân tạo
- …

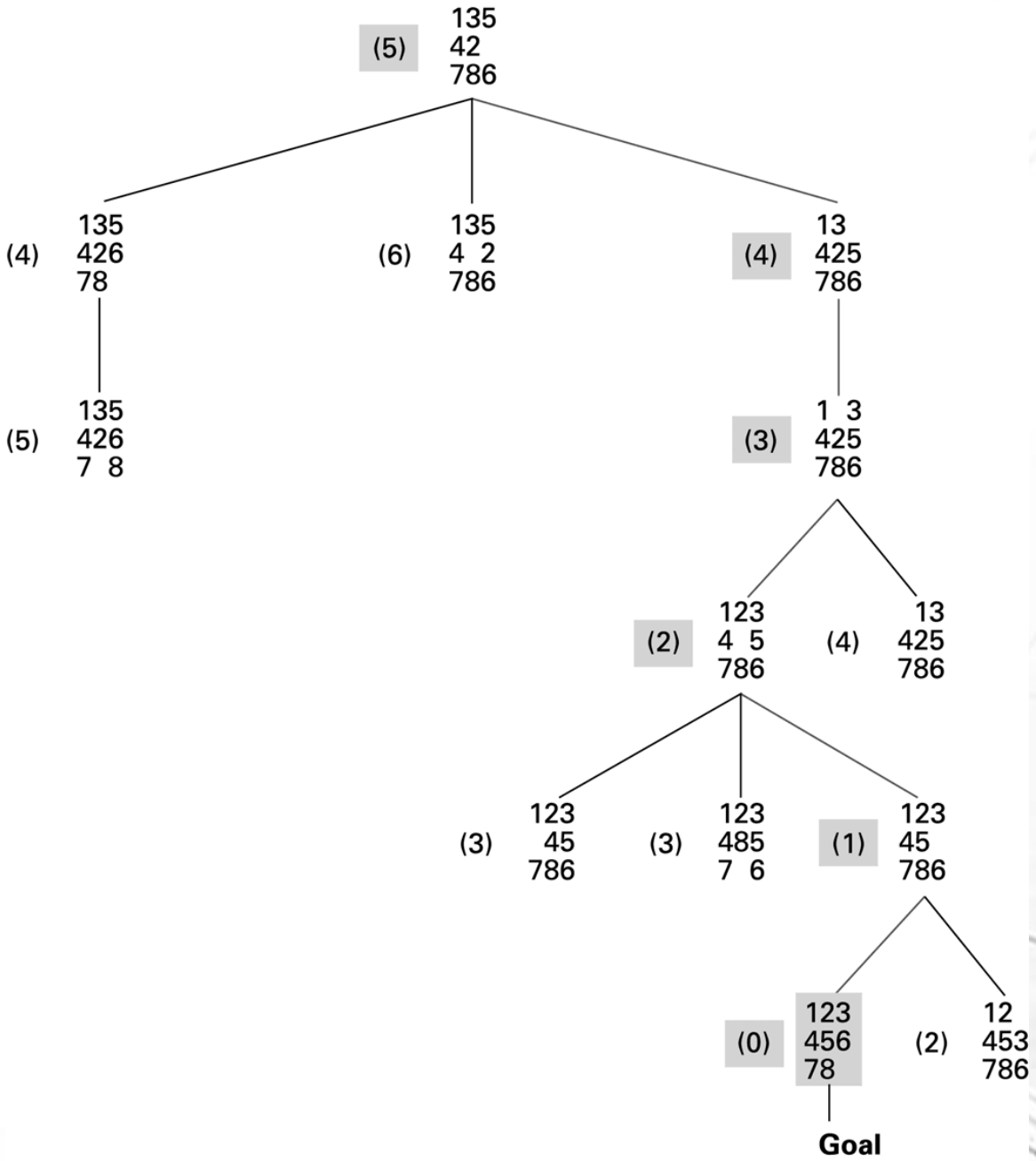
Ứng dụng



Ứng dụng


1 3 5
4 2
7 8 6
1 3 5
4 2 6
7 8
(4)
1 3 5
4 2
7 8 6
(6)
1 3
4 2 5
7 8 6
(4)
Heuristic values

Ứng dụng


(5)
135
42
786
(4)
135
426
78
(6)
135
4 2
786
(4)
13
425
786
(5)
135
426
7 8
(3)
1 3
425
786
(2)
123
4 5
786
(4)
13
425
786
(3)
123
45
786
(3)
123
485
7 6
(1)
123
45
786
(0)
123
456
78
(2)
12
453
786
Goal

Thư viện chuẩn

Thư viện STL

- `std::queue`
- `std::deque`
- `std::priority_queue`

Thư viện STL

- `std::queue`

Phương thức	Chức năng
`empty`	Kiểm tra danh sách rỗng
`front`	Truy xuất phần tử tại đầu hàng đợi
`push`	Đẩy phần tử mới vào cuối hàng đợi
`pop`	Xóa phần tử đầu hàng đợi
...	...

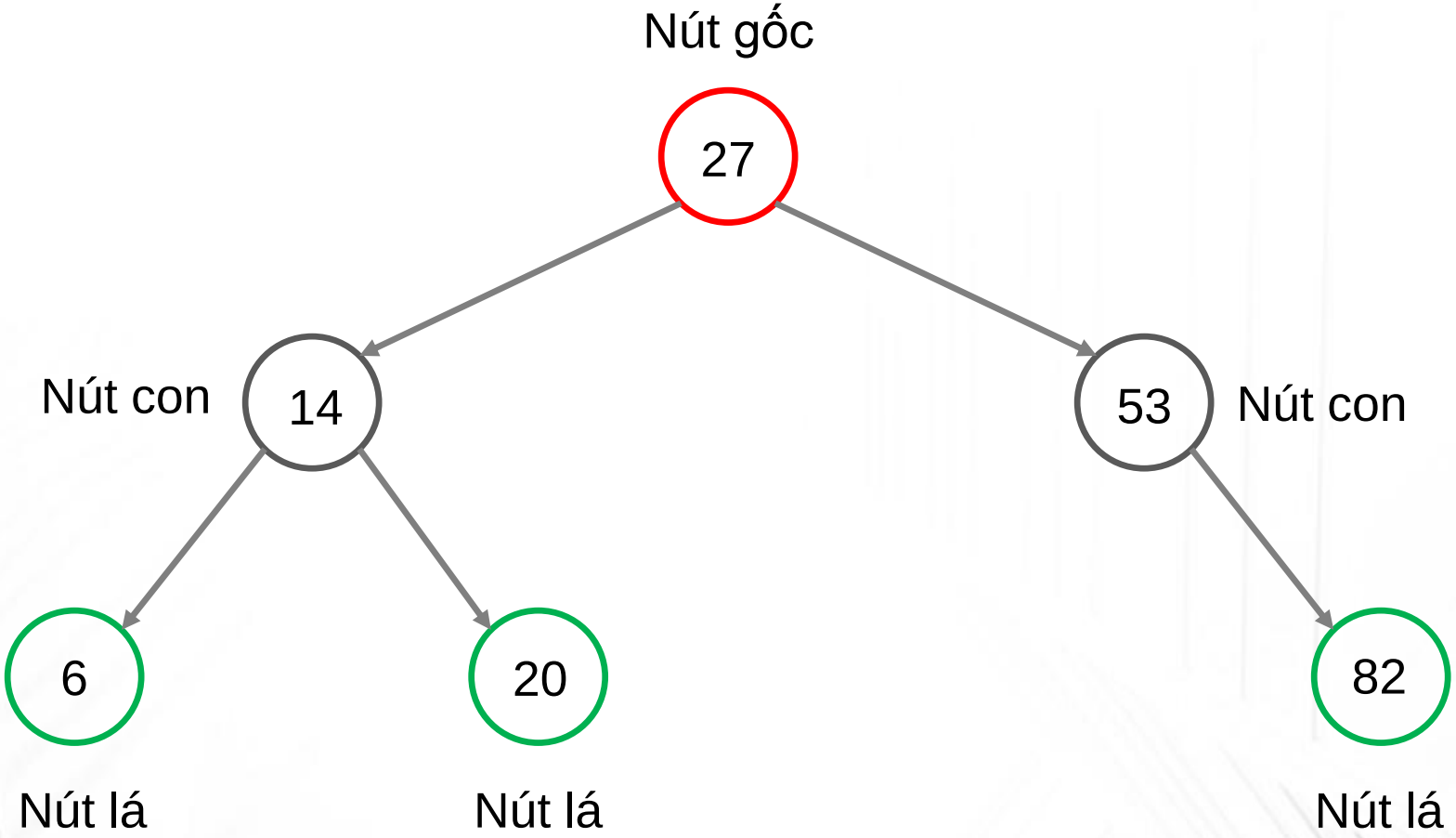
Nội dung

- Danh sách liên kết
- Ngăn xếp
- Hàng đợi
- Cây nhị phân tìm kiếm
- Các cấu trúc dữ liệu khác

Cây nhị phân tìm kiếm

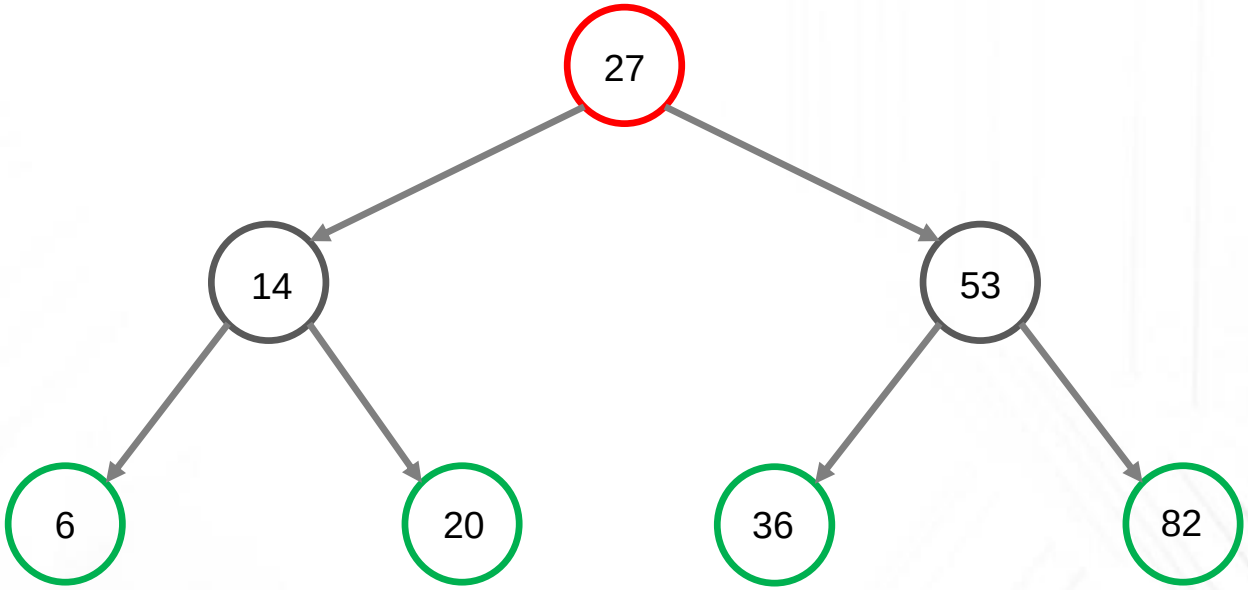
- Binary Search Tree
- Là cấu trúc dữ liệu dạng container với các nút được tổ chức theo kiểu phân cấp, **hỗ trợ tìm kiếm**
- Phân loại nút:
  - Nút gốc: không có cha
  - Nút con: có duy nhất 1 cha, có nhiều nhất 2 con
  - Nút lá: không có con

Cây nhị phân tìm kiếm



Hỗ trợ tìm kiếm

- Tại 1 nút:
  - Khóa(cây con trái) < khóa(nút hiện tại) < khóa(cây con phải)
  - Không tồn tại 2 nút cùng giá trị khóa



Cây nhị phân tìm kiếm

Định nghĩa cấu trúc dữ liệu:

```
struct TNode
{
    int key;
    TNode* pLeft;
    TNode* pRight;
};

struct BSTree
{
    TNode* root;
};
```

Cây nhị phân tìm kiếm

Các thao tác

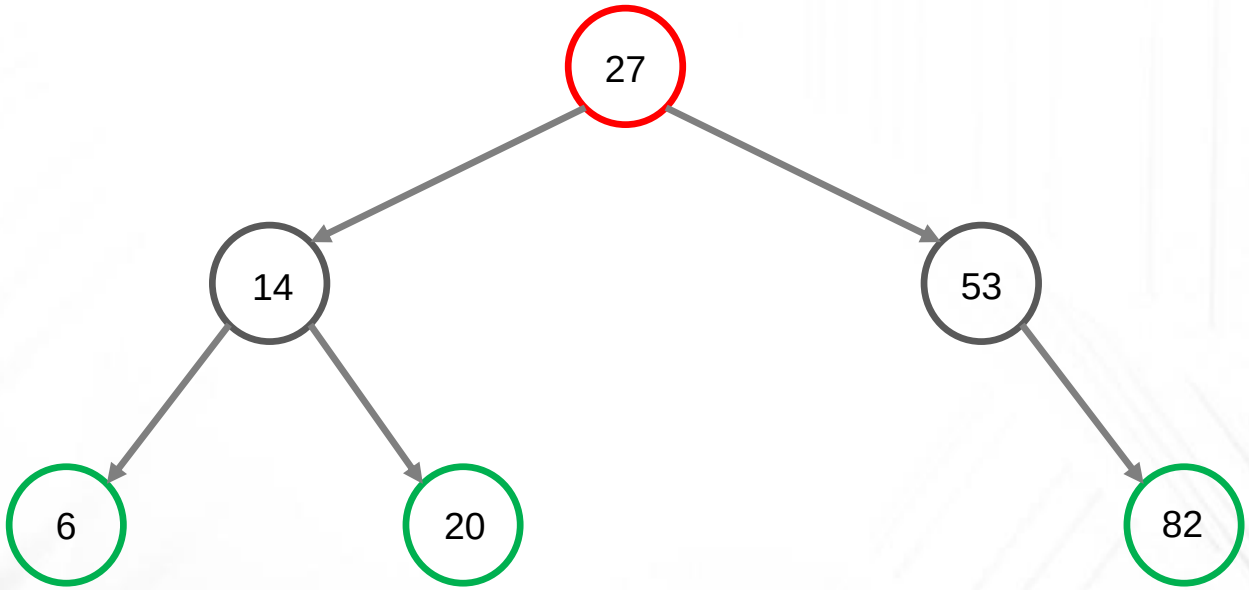
- Khởi tạo cây rỗng
- Thêm phần tử
- Tìm phần tử
- Xóa phần tử
- Duyệt cây
  - Tiền thứ tự (pre-order)
  - Trung thứ tự (in-order)
  - Hậu thứ tự (post-order)
  - Theo tầng
- Hủy cây

Cây nhị phân tìm kiếm

```
void initialize(BSTree** tree);

void insert(BSTree* tree, int value);

TNode* search(const BSTree* tree, int value);

void delete(BSTree* tree, TNode* node);

void inTravel(const BSTree* tree);

void preTravel(const BSTree* tree);

void postTravel(const BSTree* tree);

void deinitialize(BSTree** tree);
```

Thêm phần tử

- Phần tử mới luôn được thêm ở lá
- Bỏ qua nếu trùng khóa
- Ví dụ:
  - Thêm 30, 90 và 16

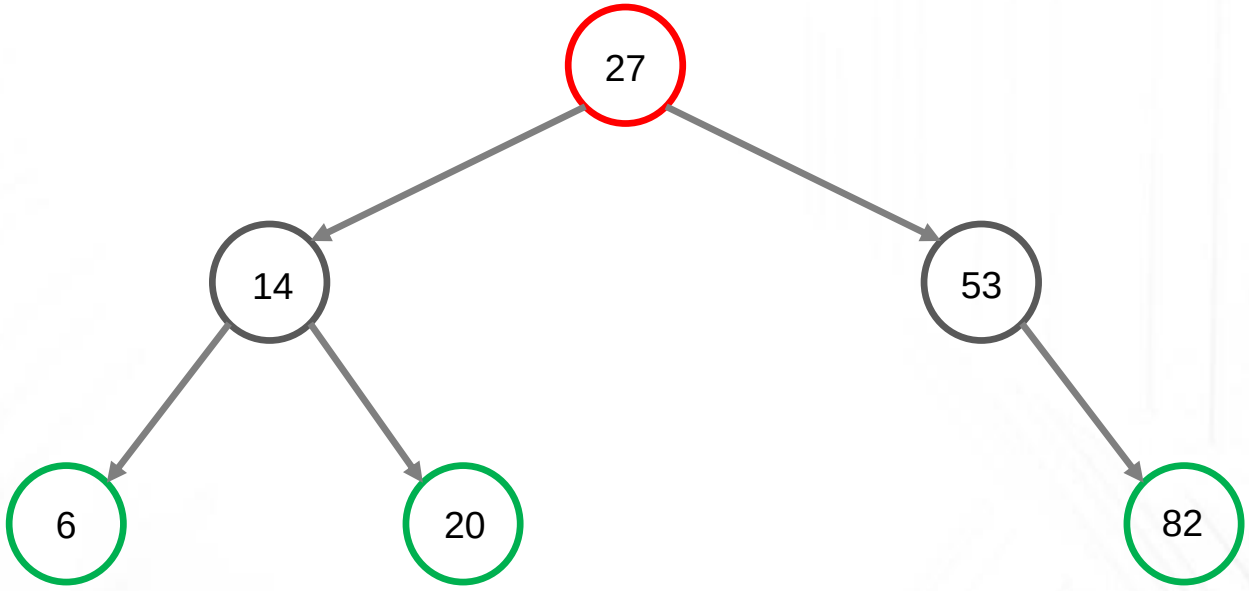


Thêm phần tử

```
void insertNode(TNode*& w_root, int r_val)
{
   nếu w_root = null thì
      w_root = newNode(r_val);
      kết thúc
   cuối nếu

   nếu w_root->key = r_val thì
     kết thúc
   cuối nếu

   nếu w_root->key < r_val thì
        insertNode(w_root->pLeft, r_val)
   ngược lại
        insertNode(w_root->pRight, r_val)
   cuối nếu
}

void insert(BSTree* u_tree, int r_val)
{
   gọi insertNode(&u_tree->root, r_val);
}
```

Duyệt cây

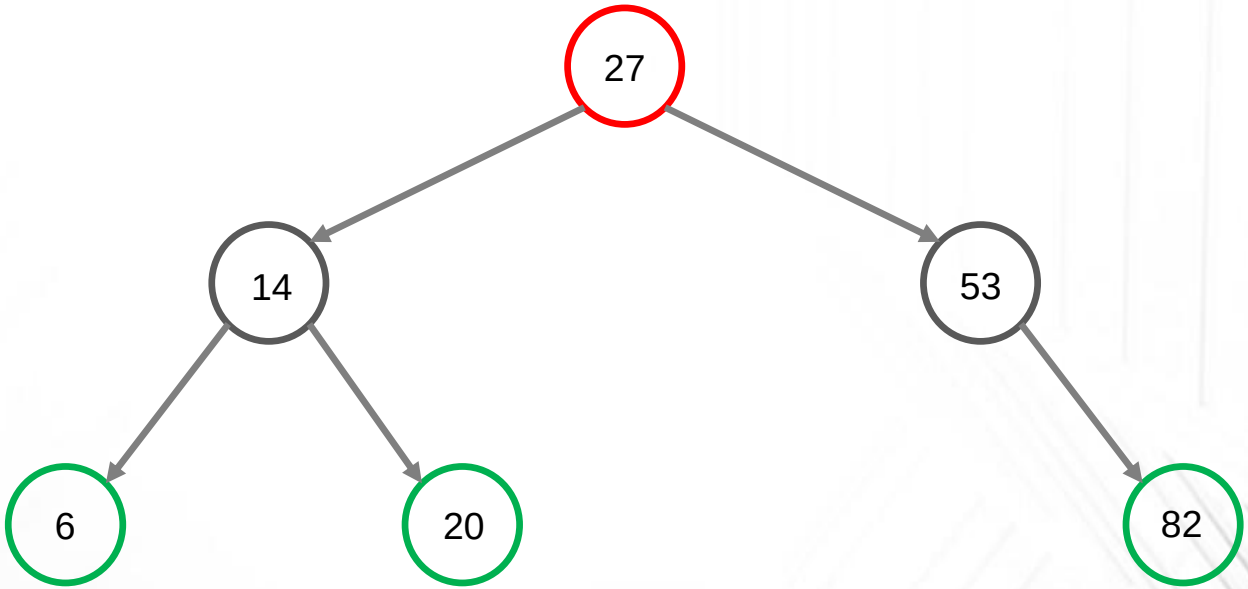
- Tiền thứ tự
  - Left – Node – Right
  - Duyệt theo thứ tự cây con trái trước, nút gốc và cây con phải



Thứ tự: 6, 14, 20, 27, 53, 82

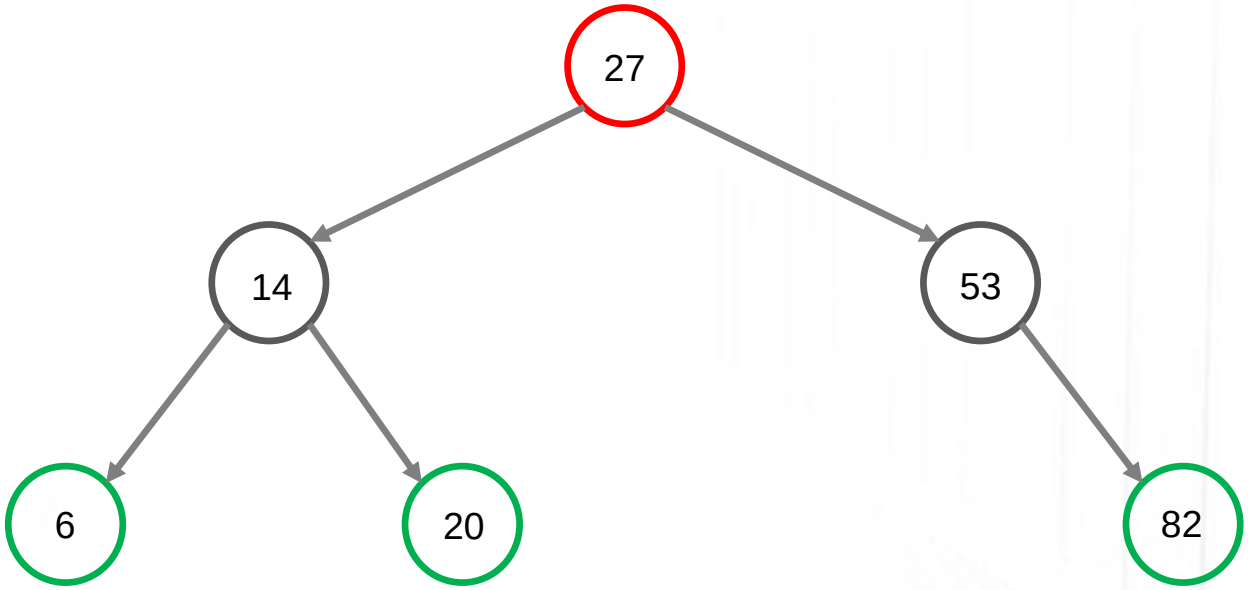
Duyệt cây

- Trung thứ tự
  - Node – Left – Right
- Hậu thứ tự
  - Left – Right – Node



Duyệt cây

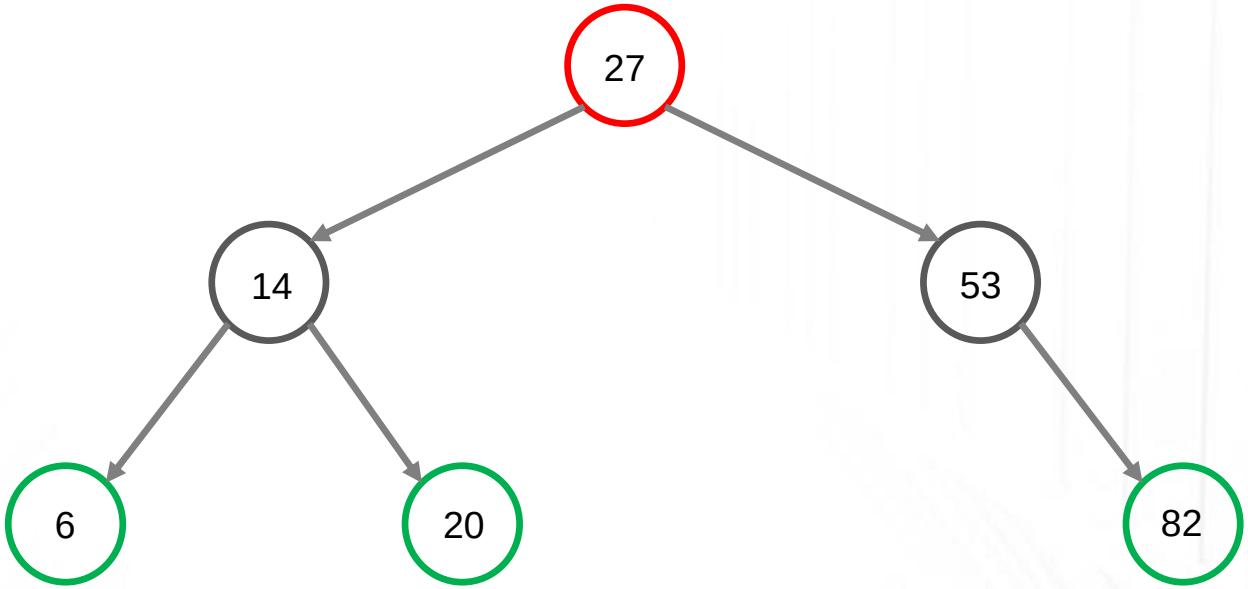
- Theo tầng



Thứ tự: 27, 14, 53, 6, 20, 82

Tìm kiếm

- Theo chiều sâu
- Theo chiều rộng



Tìm: 14, 26, 40

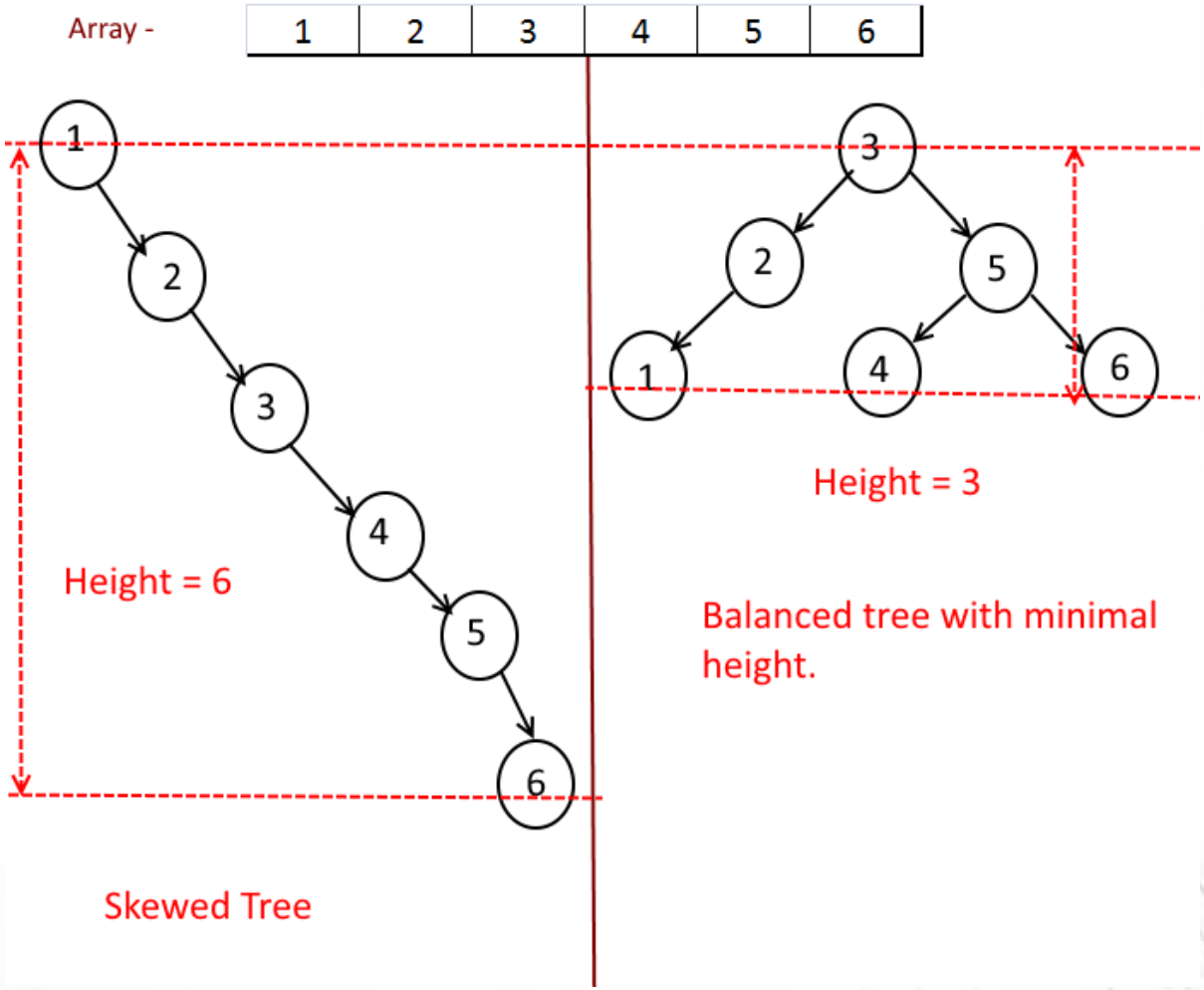
Tìm kiếm theo chiều sâu

```
TNode* depthFirstSearch(const TNode* node, int value)
{
   nếu node = null thì
      trả về null   // không tìm thấy nút thỏa điều kiện
   cuối nếu

   nếu node->key = value thì
     trả về node
   cuối nếu

   nếu node->key < value thì
      trả về depthFirstSearch(node->pLeft, value)
   cuối nếu

   trả về depthFirstSearch(node->pRight, value)
}

TNode* search(const BSTree* r_tree, int value)
{
    trả về depthFirstSearch(r_tree->root, value);
}
```

Tìm kiếm theo chiều rộng

```
TNode* breadthFirstSearch(const TNode* node, int value)
{
   ...
}

TNode* search(const BSTree* r_tree, int value)
{
    trả về breadthFirstSearch(r_tree->root, value);
}
```

Cây suy biến



https://algorithms.tutorialhorizon.com/sorted-array-to-binary-search-tree-of-minimal-height/ (09/2019)

Cây cân bằng

- Cây AVL
- Cây đỏ - đen (cây 2-3-4)
- Cây 2-3
- Skip list
- B-tree

Ứng dụng

- Cài đặt hàng đợi có độ ưu tiên
- Tìm kiếm
- Sắp xếp động
- Đánh chỉ mục trong cơ sở dữ liệu
- Tính biểu thức số học
- Quản lý bộ nhớ ảo
- …

Nội dung

- Danh sách liên kết
- Ngăn xếp
- Hàng đợi
- Cây nhị phân tìm kiếm
- Các cấu trúc dữ liệu khác

Thư viện STL

- `std::set`: tập hợp
- `std::map`: thường được cài bằng cây đỏ đen
- `std::unordered_map`:bảng băm

Đánh giá đạt mục tiêu

Sau buổi học, liệu sinh viên có thể:

- Cài đặt và sử dụng danh sách liên kết ?
- Cài đặt và sử dụng ngăn xếp ?
- Cài đặt và sử dụng hàng đợi ?
- Liệt kê các thao tác cơ bản trên cây nhị phân tìm kiếm ?