Mục lục

Buổi 1

ÔN TẬP KIẾN THỨC

1.1 Thông tin chung

Mục tiêu thực hành:

- Ôn tập một số kiến thức về lập trình C, C++ như: biến con trỏ, 3 cách truyền tham số và kiểu dữ liệu cấu trúc *(struct)*.
- Nhắc lại kỹ thuật đệ quy.

Công cụ thực hành:

- Ngôn ngữ lập trình: C, C++.
- Môi trường lập trình: Visual C++ 2010 trở lên.

Thời gian thực hành: 3 tiết.

1.2 Nội dung thực hành

1.2.1 Biến con trỏ và cấp phát động

- Biến con trỏ *(pointer)* là biến lưu địa chỉ của một kiểu dữ liệu nào đó.
- Cú pháp

```
<Kiểu dữ liệu> *<Tên biến con trỏ>;
```

- Cấp phát động được quản lý bởi biến con trỏ. Trong ngôn ngữ C++, từ khóa *new* và *delete* được dùng để *(cấp phát)* và *(hủy)* một biến con trỏ.

ex1.1

```
1 int x = 2;
2 int *p = &x;
3 cout << "p = " << p << endl;
4 cout << "&x = " << &x << endl;
```

```
5  cout << "*p = " << *p << endl;
6  cout << "&*p = " << &*p << endl;
7  cout << "*&p = " << *&p << endl;
8
9  //
10 int y = 7;
11 int *q = &y;
12 y++;
13 cout << "y = " << y << endl;
14 q++;
15 cout << "q = " << q << ", "<< "*q = " << *q << endl; // ?
16
17 //
18 int *p1 = new int;
19 *p1 = 200;
20 delete p1;
21 *p1 = 400; // error!!!
22
23 int z = 100;
24 int *p2 = new int;
25 p2 = &z;
26 delete p2; // error!!!
27
28 //
29 int *a;
30 a = new int[100];
31 for (int i = 0; i < 100; i++)
32 {
33    a[i] = i;
34    cout << a[i] << ", ";
35 }
36 delete[] a;
```

1.2.2 Truyền tham số

Có 3 cách truyền tham số:

- Truyền tham trị: thực hiện xong hàm, tham số *không thay đổi* giá trị.
- Truyền tham biến: thực hiện xong hàm, tham số *thay đổi* giá trị.
- Truyền con trỏ: thực hiện xong hàm, tham số *thay đổi* giá trị.

1.2.3 Kiểu dữ liệu cấu trúc

Ví dụ 1.2.1. *Định nghĩa cấu trúc dữ liệu của một điểm (Point) trong tọa độ Oxy với 2 thuộc tính x và y. Cài đặt hàm di chuyển tọa độ một điểm với nx là khoảng cách theo trục Ox và ny là khoảng cách theo trục Oy.*

ex1.2

```
struct Point
{
    int x;
    int y;
};
//
void MovePoint(Point &p, int nx, int ny)
{
    p.x += nx;
    p.y += ny;
}
void MovePoint(Point *&p, int nx, int ny)
{
    p->x += nx;
    p->y += ny;
}
//
int main()
{
    // ...
    Point p1;
    p1.x = 100;
    p1.y = 200;
    Point *p2 = new Point;
    p2->x = p1.x;
    p2->y = p1.x;
    MovePoint(p1, 50, 50);  //
    MovePoint(p2, 100, 100);//
}
```

1.2.4 Đệ quy

Ví dụ 1.2.2. *Cài đặt hàm đệ quy tính tổng của n số nguyên dương đầu tiên*

$$S(n) = 1 + 2 + 3 + \cdots + n - 1 + n, n > 0.$$

Lời giải. Gọi $f(n)$ là tổng của n số nguyên dương đầu tiên, công thức truy hồi tính $f(n)$ được định nghĩa như sau:

$$f(n) = \begin{cases} n & , n = 1 \\ f(n-1) + n & , n > 1 \end{cases}$$

Ví dụ 1.2.3. *Bài toán nuôi thỏ (dãy Fibonacci)*

Lời giải. Gọi $f(n)$ là số cặp thỏ ở tháng thứ n, áp dụng công thức truy hồi của dãy Fibonacci để tính $f(n)$:

$$f(n) = \begin{cases} 1 & , n = 0, 1 \\ f(n-1) + f(n-2) & , n > 1 \end{cases}$$

1.3 Bài tập thực hành

1. Định nghĩa cấu trúc dữ liệu phân số với 2 thuộc tính: tử số và mẫu số. Cài đặt các hàm tính tổng, hiệu, tích, thương, rút gọn phân số.
2. Cài đặt hàm đệ quy đổi số nguyên dương n sang hệ nhị phân.
3. Cài đặt hàm đệ quy tính ước số chung lớn nhất của 2 số nguyên dương a và b.

Buổi 2

THUẬT TOÁN TÌM KIẾM

2.1 Thông tin chung

Mục tiêu thực hành:

- Ôn tập kiến thức về đọc, ghi dữ liệu từ tập tin văn bản *(*.txt)*.
- Hướng dẫn cài đặt 2 thuật toán tìm kiếm tuyến tính và tìm kiếm nhị phân.

Công cụ thực hành:

- Ngôn ngữ lập trình: C, C++.
- Môi trường lập trình: Visual C++ 2010 trở lên.

Thời gian thực hành: 3 tiết.

2.2 Nội dung thực hành

2.2.1 Đọc, ghi dữ liệu từ tập tin

- Thêm thư viện $fstream$ vào chương trình.
- Khai báo kiểu dữ liệu $ifstream, ofstream$ và sử dụng 2 toán tử $\gg, \ll$ để đọc, ghi dữ liệu từ tập tin.

Ví dụ 2.2.1. *Hàm đọc và ghi mảng 1 chiều từ tập tin* `*.txt`

```
1 void ReadFile(const char *filename, int *&a, int &n)
2 {
3    ifstream file;
4    file.open(filename);
5    if (file)
6    {
7       file >> n;
8       a = new int[n];
9       for (int i = 0; i < n; i++)
```

```
10          file >> a[i];
11      }
12      file.close();
13  }
14
15  void WriteFile(const char *filename, int *a, int n)
16  {
17      ofstream file;
18      file.open(filename);
19      if (file)
20      {
21          file << n << endl;
22          for (int i = 0; i < n; i++)
23              file << a[i] << " ";
24      }
25      file.close();
26  }
```

2.2.2 Thuật toán tìm kiếm tuyến tính

Ví dụ 2.2.2. *Hàm tìm kiếm tuyến tính có sử dụng phần tử lính canh.*

2.2.3 Thuật toán tìm kiếm nhị phân

Ví dụ 2.2.3. *Hàm tìm kiếm nhị phân có sử dụng đệ quy.*

Ví dụ 2.2.4. *Xây dựng thuật toán tìm kiếm tam phân dựa trên thuật toán tìm kiếm nhị phân.*

Lời giải. Khai báo 2 phần tử giữa là $mid1$ và $mid2$, mỗi lần thực hiện chia mảng thành 3 mảng con:

$$a_0, \ldots, a_{mid1}, \cdots, a_{mid2}, \cdots a_{n-1}$$

Gọi đệ quy thuật toán thực hiện với 1 trong 3 mảng con.

```
1  int TernarySearch(int *a, int left, int right, int x)
2  {
3      int mid1, mid2;
4      if (left > right)
5          return -1;
6
7      mid1 = (left + right) / 3;
8      if (a[mid1] == x)
9          return mid1;
```

```
10      mid2 = mid1 + (left + right) / 3;
11      if (a[mid2] == x)
12          return mid2;
13
14      if (a[mid1] > x)
15          return TernarySearch(a, left, mid1 - 1, x);
16      else if (a[mid2] < x)
17          return TernarySearch(a, mid2 + 1, right, x);
18      else // a[mid1] < x < a[mid2]
19          return TernarySearch(a, mid1 + 1, mid2 - 1, x);
20  }
```

2.3 Bài tập thực hành

1. Áp dụng kỹ thuật lính canh, cài đặt hàm tìm số có giá trị lớn nhất trong mảng 1 chiều.
2. Cài đặt hàm tìm các số nguyên tố nằm trong mảng 1 chiều có n phần tử.
3. Cài đặt lại hàm TernarySearch không sử dụng đệ quy.

Buổi 3

THUẬT TOÁN SẮP XẾP

3.1 Thông tin chung

Mục tiêu thực hành:

- Hướng dẫn cài đặt một số thuật toán sắp xếp cơ bản như: sắp xếp chọn, sắp xếp nổi bọt, ...

Công cụ thực hành:

- Ngôn ngữ lập trình: C, C++.
- Môi trường lập trình: Visual C++ 2010 trở lên.

Thời gian thực hành: 3 tiết.

3.2 Nội dung thực hành

3.2.1 Sắp xếp nổi bọt

Ví dụ 3.2.1. *Hàm cài đặt thuật toán sắp xếp nổi bọt BubbleSort()*

```
1 void BubbleSort(int *a, int n)
2 {
3    int i, j;
4    for (i = 0; i < n - 1; i++)
5       for (j = n - 1; j > i; j--)
6          if (a[j] < a[j - 1])
7             Swap(a[j], a[j - 1]);
8 }
```

Ví dụ 3.2.2. *Bài toán di chuyển đĩa*

- *Cho một dãy đĩa gồm 2 màu sáng và tối*

- *Cài đặt thuật toán di chuyển các đĩa sáng về bên trái, đĩa tối về bên phải và đếm số hoán vị hai đĩa khi thực hiện. Mỗi bước chỉ được hoán vị hai đĩa kề nhau.*

Gợi ý: xem dãy đĩa như một dãy số, đĩa màu sáng có giá trị là 0 và đĩa tối có giá trị là 1.

Lời giải. Áp dụng giải thuật sắp xếp nổi bọt và một cờ để đánh dấu sau khi một cặp đĩa được hoán vị. □

```
void BubbleSortEx(int *a, int n)
{
    bool swapped;
    int i, j;
    for (i = 0; i < n - 1; i++)
    {
        swapped = false;
        for (j = n - 1; j > i; j--)
        {
            if (a[j] < a[j - 1])
            {
                Swap(a[j], a[j - 1]);
                swapped = true;
            }
        }
        if (swapped == false)
            break;
    }
}
```

3.2.2 Sinh ngẫu nhiên dữ liệu

- Thêm thư viện *ctime* vào chương trình.
- Sử dụng hàm *rand*() sinh ngẫu nhiên một số.

Ví dụ 3.2.3. *Hàm sinh ngẫu nhiên mảng 1 chiều gồm n phần tử.*

```
void RandArray(int *&a, int &n)
{
    // ...
    srand((unsigned) time(NULL)); // ?
    for (int i = 0; i < n; i++)
        a[i] = rand();
}
```

3.2.3 Tính thời gian thực hiện

- Trong hàm *main*, thêm vào đoạn chương trình sau để tính thời gian thực hiện.
- Khai báo 2 biến kiểu *double* tương ứng thời gian trước và sau khi thực hiện một hàm.

Ví dụ 3.2.4. *Tính thời gian thực hiện của một thuật toán.*

```
int main()
{
    int n;
    int *a;
    RandArray(a, n);

    const char *file = "data.txt";
    WriteFile(file, a, n);

    double t1, t2; // Thoi gian bat dau->ket thuc

    t1 = clock();
    BubbleSort(a, n);
    t2 = clock();
    cout << "BubbleSort: " << (t2 - t1) / CLK_TCK;

    ReadFile(file, a, n);

    t1 = clock();
    SelectionSort(a, n);
    t2= clock();
    cout << "SelectionSort: " << (t2 - t1) / CLK_TCK;

    return 0;
}
```

3.3 Bài tập thực hành

1. Cài đặt hàm tương ứng với các thuật toán sắp xếp cơ bản.
2. Cài đặt hàm đếm số phép so sánh và hoán vị của thuật toán sắp xếp cơ bản trong trường hợp tốt nhất *(mảng đã có thứ tự)*.

Buổi 4

THUẬT TOÁN SẮP XẾP (tt)

4.1 Thông tin chung

Mục tiêu thực hành:

- Hướng dẫn cài đặt thuật toán sắp xếp nhanh *(QuickSort)*, sắp xếp trộn *(MergeSort)*, sắp xếp vun đống *(HeapSort)*.

Công cụ thực hành:

- Ngôn ngữ lập trình: C, C++.
- Môi trường lập trình: Visual C++ 2010 trở lên.

Thời gian thực hành: 3 tiết.

4.2 Nội dung thực hành

4.2.1 Thuật toán sắp xếp nhanh

Ví dụ 4.2.1. *Hàm phân hoạch mảng Partition() và thuật toán sắp xếp nhanh QuickSort().*

```
1  void Partition(int *a, int &i, int &j, int x)
2  {
3     do
4     {
5        while (a[i] < x)
6           i++;
7        while (a[j] > x)
8           j--;
9        if (i <= j)
10       {
11          cout << "i = " << i << ", j = " << j << endl;
12          Swap(a[i], a[j]);
13          i++;
```

```
14            j--;
15        }
16    } while (i < j);
17 }
18
19 void QuickSort(int *a, int left, int right)
20 {
21    int i, j, x;
22    x = a[(left + right) / 2];
23    i = left;
24    j = right;
25
26    Partition(a, i, j, x);
27    if (left < j)
28       QuickSort(a, left, j);
29    if (i < right)
30       QuickSort(a, i, right);
31 }
```

4.2.2 Thuật toán sắp xếp trộn

Ví dụ 4.2.2. *Hàm trộn 2 mảng* $Merge()$ *và thuật toán sắp xếp trộn* $MergeSort()$.

```
1  void Merge(int *a, int left, int middle, int right)
2  {
3     int i, j, k;
4     int *b;
5     i = left;
6     j = middle + 1;
7     k = left;
8     b = new int[(right - left + 1)];
9
10    while (i <= middle && j <= right)
11    {
12       if (a[i] <= a[j])
13          b[k++] = a[i++];
14       else
15          b[k++] = a[j++];
16    }
17    while (i <= middle)
18       b[k++] = a[i++];
19    while (j <= right)
20       b[k++] = a[j++];
```

```
21     for (k = left; k <= right; k++)
22        a[k] = b[k];
23  }
24
25  void MergeSort(int *a, int left, int right)
26  {
27     int middle;
28     if (left < right)
29     {
30        middle = (left + right) / 2;
31        MergeSort(a, left, middle);
32        MergeSort(a, middle + 1, right);
33        Merge(a, left, middle, right);
34     }
35  }
```

4.2.3 Thuật toán sắp xếp vun đống

Ví dụ 4.2.3. *Hàm tạo* $MakeHeap()$*, hiệu chỉnh* $Heapify()$ *và sắp xếp vun đống* $HeapSort()$*.*

```
1   void Heapify(int *a, int left, int right)
2   {
3      // j = 2 * i
4      //           a[i]
5      //     -------- --------
6      //     |               |
7      //     a[j]            a[j + 1]
8      int i, j, x;
9      i = left;
10     j = 2 * i + 1;
11     x = a[i];
12     while (j < right)
13     {
14        if (j < right)
15           if (a[j] < a[j + 1])
16              j++;
17        if (a[j] < x)
18           return;
19        else
20        {
21           a[i] = a[j];
22           a[j] = x;
```

```
            i = j;
            j = 2 * i + 1;
        }
    }
}

void MakeHeap(int *a, int n)
{
    int left;
    left = n / 2;
    while (left >= 0)
    {
        Heapify(a, left, n);
        left--;
    }
}

void HeapSort(int *a, int n)
{
    int right = n - 1;
    MakeHeap(a, n);
    while (right > 0)
    {
        Swap(a[0], a[right]);
        right--;
        Heapify(a, 0, right);
    }
}
```

4.3 Bài tập thực hành

1. Cài đặt thuật toán QuickSort với x là phần tử đầu dãy. Khi dãy đã có thứ tự, so sánh thời gian thực hiện so với trường hợp x là phần tử giữa của dãy.
2. Cài đặt thuật toán QuickSort với x là phần tử median of three *(trung vị của 3 phần tử: đầu, giữa, cuối dãy)*.
3. Xây dựng thuật toán MergeSort trộn 3 dãy sắp xếp theo thứ tự tăng dần.

Buổi 5

DANH SÁCH LIÊN KẾT ĐƠN

5.1 Thông tin chung

Mục tiêu thực hành:

- Giới thiệu danh sách liên kết đơn, định nghĩa cấu trúc dữ liệu của một nút và một danh sách.
- Cài đặt các thao tác trên danh sách liên kết đơn như: thêm, xóa, tìm kiếm, ...

Công cụ thực hành:

- Ngôn ngữ lập trình: C, C++.
- Môi trường lập trình: Visual C++ 2010 trở lên.

Thời gian thực hành: 3 tiết.

5.2 Nội dung thực hành

5.2.1 Cấu trúc một nút trong danh sách liên kết

Ví dụ 5.2.1. *Cấu trúc của một nút gồm: thành phần dữ liệu (Info) và thành phần liên kết (con trỏ pNext).*

```
1 typedef int Data;
2 const Data NULL_DATA = -1;
3
4 struct Node
5 {
6    Data Info; // Data: Point, Student, ...
7    Node *pNext;
8 };
```

5.2.2 Cấu trúc một danh sách liên kết đơn

Ví dụ 5.2.2. *Đối với danh sách liên kết đơn, cấu trúc dữ liệu gồm 2 con trỏ: pHead và pTail (hay chỉ cần pHead cũng được)*

```
struct List
{
    Node *pHead;
    Node *pTail;
};
```

5.2.3 Các thao tác trên danh sách liên kết đơn

Ví dụ 5.2.3. *Các hàm khởi tạo một nút và danh sách liên kết đơn.*

```
Node *InitNode(Data x)
{
    Node *p = new Node();
    if (p == NULL)
        return NULL;
    p->Info = x;
    p->pNext = NULL;
    return p;
}

void InitList(List *l)
{
    l->pHead = NULL;
    l->pTail = NULL;
}
```

Ví dụ 5.2.4. *Các hàm cài đặt những thao tác cơ bản như: thêm, xóa, in, ... danh sách liên kết đơn.*

```
void InsertHead(List *l, Student x)
{
    Node *p = InitNode(x);
    if (l->pHead == NULL)
    {
        l->pHead = p;
        l->pTail = p;
    }
    else
    {
        p->pNext = l->pHead;
```

```
12          l->pHead = p;
13      }
14  }
15
16  void InsertAfter(List *l, Node *q, Data x)
17  {
18      Node *p = InitNode(x);
19
20      if (q != NULL)
21      {
22          p->pNext = q->pNext;
23          q->pNext = p;
24          if (q == l->pTail)
25              l->pTail = p;
26      }
27  }
28
29  void RemoveHead(List *l)
30  {
31      Node *p = NULL;
32      if (l->pHead != NULL) // Danh sach khac rong
33      {
34          p = l->pHead;
35          l->pHead = l->pHead->pNext;
36          delete p;
37          if (l->pHead == NULL)
38              l->pTail = NULL;
39      }
40  }
41
42  void RemoveTail(List *l)
43  {
44      Node *p = NULL;
45      if (l->pHead != NULL) // TH1. Danh sach khac rong
46      {
47          p = l->pHead;
48          if (p == l->pTail) // TH2: 1 nut
49          {
50              delete p;
51              l->pHead = NULL;
52              l->pTail = NULL;
53              return;
```

```
54          }
55          // TH3: > 2 nut
56          while (p->pNext != l->pTail)
57              p = p->pNext;
58          l->pTail = p;
59          l->pTail->pNext = NULL;
60          p = p->pNext;
61          delete p;
62      }
63  }
64
65  // ...
66  void PrintList(const List *l)
67  {
68      Node *p = l->pHead;
69
70      while (p != NULL)
71      {
72          cout << p->Info << "->";
73          p = p->pNext;
74      }
75  }
```

5.3 Bài tập thực hành

1. Định nghĩa cấu trúc dữ liệu của danh sách liên kết đôi.
2. Cài đặt các thao tác cơ bản đối với danh sách liên kết đôi.

Buổi 6

ỨNG DỤNG CỦA DANH SÁCH LIÊN KẾT ĐƠN

6.1 Thông tin chung

Mục tiêu thực hành:

- Định nghĩa cấu trúc dữ liệu của danh sách liên kết đơn quản lý thông tin sinh viên gồm: mã số, họ tên, điểm trung bình. Cài đặt các thao tác trên danh sách sinh viên như: thêm, xóa, tìm kiếm.
- Sắp xếp danh sách theo thứ tự giảm dần của điểm trung bình.

Công cụ thực hành:

- Ngôn ngữ lập trình: C, C++.
- Môi trường lập trình: Visual C++ 2010 trở lên.

Thời gian thực hành: 3 tiết.

6.2 Nội dung thực hành

6.2.1 Cấu trúc một nút trong danh sách liên kết

Ví dụ 6.2.1. *Cấu trúc của một nút gồm: thành phần dữ liệu (Info) và thành phần liên kết (con trỏ pNext).*

```
1 #include "string"
2
3 struct Student
4 {
5     string    Id;
6     string    Fullname;
7     float     Score;
8 };
```

```
9
10 struct Node
11 {
12     Student Info;
13     Node  *pNext;
14 };
```

6.2.2 Cấu trúc một danh sách liên kết đơn

Ví dụ 6.2.2. *Cấu trúc dữ liệu danh sách sinh viên gồm 2 con trỏ: pHead và pTail.*

```
1 struct List
2 {
3     // ...
4 };
```

6.2.3 Các thao tác trên danh sách liên kết đơn

Ví dụ 6.2.3. *Các hàm khởi tạo một nút và danh sách sinh viên.*

```
1 Node *InitNode(Student x)
2 {
3     // ...
4 }
5
6 void InitList(List *l)
7 {
8     // ...
9 }
```

Ví dụ 6.2.4. *Các hàm khởi tạo một nút và danh sách sinh viên.*

```
1 void InsertHead(List *l, Student x)
2 {
3     // ...
4 }
5
6 void RemoveHead(List *l)
7 {
8     // ...
9 }
10
11 void PrintList(const List *l)
12 {
```

```
13      Node *p = l->pHead;
14      while (p != NULL)
15      {
16          cout << "[";
17          cout << p->Info.Id << ", ";
18          cout << p->Info.Fullname << ", ";
19          cout << p->Info.Score;
20          cout << "]->";
21          p = p->pNext;
22      }
23  }
```

6.2.4 Sắp xếp danh sách sinh viên

Ví dụ 6.2.5. *Hàm cài đặt thuật toán sắp xếp chọn trên danh sách sinh viên theo thứ tự giảm dần của điểm số.*

```
1   void ListSelectionSort(List *l)
2   {
3       Node *p, *q, *max;
4       p = l->pHead;
5       while (p != l->pTail)
6       {
7           q = p->pNext;
8           max = p;
9           while (q != NULL)
10          {
11              if (q->Info.Score > max->Info.Score)
12                  max = q;
13              q = q->pNext;
14          }
15          Swap(p->Info, max->Info);
16          p = p->pNext;
17      }
18  }
```

Ví dụ 6.2.6. *Hàm cài đặt thuật toán sắp xếp nhanh (QuickSort) trên danh sách liên kết.*

```
1   void ListAppend(List *l, List *l1, Node *x, List *l2)
2   {
3       if (l1->pHead == NULL)
4           l->pHead = x;
5       else
```

```
6     {
7        l->pHead = l1->pHead;
8        l1->pTail->pNext = x;
9     }
10    x->pNext = l2->pHead;
11    if (l2->pHead == NULL)
12       l->pTail = x;
13    else
14       l->pTail = l2->pTail;
15 }
16
17 void ListQuickSort(List *l)
18 {
19    Node *x, *p;
20    List *l1, *l2;
21
22    if (l->pHead == l->pTail)
23       return;
24
25    // Khoi tao Node, List ...
26
27    x = l->pHead;
28    l->pHead = x->pNext;
29    while (l->pHead != NULL)
30    {
31       p = l->pHead;
32       l->pHead = p->pNext;
33       p->pNext = NULL;
34       if (p->Info.Score >= x->Info.Score)
35          InsertTail(l1, p);
36       else
37          InsertTail(l2, p);
38    }
39    ListQuickSort(l1);
40    ListQuickSort(l2);
41    ListAppend(l, l1, x, l2);
42 }
```

6.3 Bài tập thực hành

1. Cài đặt thuật toán sắp xếp trộn trên danh sách sinh viên.

2. Xây cấu trúc dữ liệu thích hợp để biểu diễn đa thức $P(x)$

$$P(x) = c_1 x^{e_1} + c_2 x^{e_2} + \cdots + c_n x^{e_n}$$

với c_i là hệ số và e_i là số mũ, $1 \leq i \leq n$
Cài đặt hàm thực hiện các thao tác:

- Thêm một phần tử vào cuối đa thức.
- In danh sách các phần tử.
- Tính giá trị đa thức với x cho trước.

Buổi 7

NGĂN XẾP

7.1 Thông tin chung

Mục tiêu thực hành:

- Hướng dẫn cài đặt ngăn xếp bằng mảng và danh sách liên kết đơn.
- Áp dụng ngăn xếp khử đệ quy.

Công cụ thực hành:

- Ngôn ngữ lập trình: C, C++.
- Môi trường lập trình: Visual C++ 2010 trở lên.

Thời gian thực hành: 3 tiết.

7.2 Nội dung thực hành

7.2.1 Cài đặt ngăn xếp bằng mảng 1 chiều

Ví dụ 7.2.1. *Định nghĩa cấu trúc dữ liệu của một ngăn xếp và các hàm khởi tạo.*

```
1  const int MAX_SIZE = 10;
2  const int NULL_DATA = -1;
3
4  typedef int Data;
5  struct Stack
6  {
7     Data Elements[MAX_SIZE];
8     int Top;
9  };
10
11 void InitStack(Stack *s)
12 {
13    s->Top = -1;
14 }
```

Hàm kiểm tra ngăn xếp rỗng, ngăn xếp đầy.

```
bool IsEmpty(const Stack *s)
{
   return s->Top == -1;
}

bool IsFull(const Stack *s)
{
   return s->Top == (MAX_SIZE - 1);
}
```

Hàm thêm phần tử Push() và lấy phần tử Pop() từ đỉnh ngăn xếp.

```
void Push(Stack *s, Data x)
{
   if (!IsFull(s))
      s->Elements[++s->Top] = x;
   else
      cout << "Stack is full!";
}

Data Pop(Stack *s)
{
   return !IsEmpty(s) ? s->Elements[s->Top--] : NULL_DATA;
}

void PrintStack(const Stack *s)
{
   for (int i = 0; i <= s->Top; i++)
      cout << s->Elements[i] << endl;
}
```

7.2.2 Cài đặt ngăn xếp bằng danh sách liên kết

Ví dụ 7.2.2. *Định nghĩa cấu trúc dữ liệu ngăn xếp và các thao tác thêm, xóa, in, ...*

```
typedef int Data;
struct Node
{
   Data Info;
   Node *pNext;
};

struct Stack
```

```
{
    int Count;
    Node *pTop;
};

Node *InitNode(Data x)
{
    // ...
}

void InitStack(Stack *s)
{
    // ...
}

bool IsEmpty(const Stack *s)
{
    return s->pTop == NULL;
}
```

Hai thao tác cơ bản trong ngăn xếp: thêm phần tử Push() và lấy phần tử Pop() tại vị trí đỉnh của ngăn xếp.

```
void Push(Stack *s, Data x) // InsertHead
{
    Node *p = InitNode(x);

    if (IsEmpty(s) == true)
        s->pTop = p;
    else
    {
        p->pNext = s->pTop;
        s->pTop = p;
    }
    s->Count++;
}

Data Pop(Stack *s) // RemoveHead
{
    if (IsEmpty(s) == true)
        return NULL_DATA;

    Node *p = new Node();
    p = s->pTop;
```

```
22      s->pTop = s->pTop->pNext;
23      s->Count--;
24
25      Data x = p->Info;
26      delete p;
27      return x;
28  }
29
30  Data GetTop(Stack *s)
31  {
32      Node *p = new Node();
33
34      if (IsEmpty(s) == true)
35          return NULL_DATA;
36      p = s->pTop;
37
38      return p->Info;
39  }
```

7.2.3 Áp dụng ngăn xếp khử đệ quy

Ví dụ 7.2.3. *Hàm chuyển số nguyên dương trong hệ thập phân sang hệ nhị phân. Sử dụng hàm to_string() chuyển một số sang kiểu dữ liệu string.*

```
1   string DecToBin1(int n)
2   {
3       if (n == 0 || n == 1 )
4           return to_string(n);
5       return DecToBin(n / 2) + to_string(n % 2);
6   }
7
8   string DecToBin2(int n)
9   {
10      Stack *s = new Stack;
11      InitStack(s);
12      while (n > 0)
13      {
14          Push(s, n % 2);
15          n /= 2;
16      }
17
18      string binary;
19      Node * p = s->pTop;
```

```
20      while (p != NULL)
21      {
22          binary += to_string(p->Info);
23          p = p->pNext;
24      }
25      return binary;
26  }
```

7.3 Bài tập thực hành

Áp dụng ngăn xếp khử đệ quy:

1. Thuật toán tháp Hà Nội *(Ha Noi tower)*.
2. Thuật toán tìm kiếm nhị phân *(Binary Search)*.
3. Thuật toán sắp xếp nhanh *(Quick Sort)*.

Buổi 8

HÀNG ĐỢI

8.1 Thông tin chung

Mục tiêu thực hành:

- Hướng dẫn cài đặt hàng đợi bằng mảng và danh sách liên kết đơn. Áp dụng hàng đợi giải bài toán Josephus.
- Giới thiệu hàng đợi ưu tiên và hướng dẫn cài đặt.

Công cụ thực hành:

- Ngôn ngữ lập trình: C, C++.
- Môi trường lập trình: Visual C++ 2010 trở lên.

Thời gian thực hành: 3 tiết.

8.2 Nội dung thực hành

8.2.1 Cài đặt hàng đợi bằng danh sách liên kết

Ví dụ 8.2.1. *Cài đặt tương tự ngăn xếp, nhưng cấu trúc một hàng đợi gồm 2 con trỏ* $pFront, pRear$ *và các thao tác thêm cuối và lấy phần tử ở vị trí đầu hàng đợi.*

```
1  typedef int Data;
2  struct Node
3  {
4      // ...
5  };
6
7  struct Queue
8  {
9      int Count;
10     Node *pFront;
11     Node *pRear;
```

```
12 };
13
14 Node *InitNode(Data x)
15 {
16    // ...
17 }
18 void InitQueue(Queue *q)
19 {
20    // ...
21 }
22
23 bool IsEmpty(const Queue *q)
24 {
25    return q->pFront == NULL;
26 }
27
28 void EnQueue(Queue *q, Data x) // InsertTail
29 {
30    Node *p = InitNode(x);
31    if (IsEmpty(q) == true)
32    {
33       q->pFront = p;
34       q->pRear = p;
35    }
36    else
37    {
38       q->pRear->pNext = p;
39       q->pRear = p;
40    }
41    q->Count++;
42 }
43
44 Data DeQueue(Queue *q) // RemoveHead
45 {
46    if (IsEmpty(q) == true)
47       return NULL_DATA;
48
49    Node *p = new Node();
50    p = q->pFront;
51    q->pFront = q->pFront->pNext;
52    q->Count--;
53
```

```
54      Data x = p->Info;
55      delete p;
56      return x;
57  }
58
59  Data GetFront(Queue *q)
60  {
61      Node *p = new Node();
62
63      if (IsEmpty(q) == true)
64          return NULL_DATA;
65      p = q->pFront;
66
67      return p->Info;
68  }
```

Ví dụ 8.2.2. *Hàm giải bài toán Josephus.*

```
1  void Josephus(Queue *q, int m, int n)
2  {
3      for (int i = 1; i <= n; i++)
4          EnQueue(q, i);
5
6      while (!IsEmpty(q))
7      {
8          for (int j = 0; j < m - 1; j++)
9              EnQueue(q, DeQueue(q));
10         int x = DeQueue(q);
11         cout << x << "died" << endl;
12     }
13 }
```

8.2.2 Hàng đợi ưu tiên

- Hàng đợi ưu tiên (Priority Queue) tương tự như hàng đợi nhưng cấu trúc dữ liệu mỗi nút có thêm một thuộc tính quy định độ ưu tiên.
- Hai thao tác cơ bản:
  - EnQueue: thêm vào cuối hàng đợi.
  - DeleteMin/DeQueue: lấy nút có giá trị của độ ưu tiên **nhỏ nhất** *(độ ưu tiên cao nhất).*
- Cài đặt hàng đợi ưu tiên sử dụng danh sách liên kết:

– Danh sách liên kết có thứ tự: thao tác EnQueue phải tìm nút phù hợp.
– Danh sách liên kết không thứ tự: thao tác DeleteMin phải tìm nút phù hợp.

Ví dụ 8.2.3. *Tạo dự án mới và định nghĩa cấu trúc hàng đợi ưu tiên như sau:*

```
struct Node
{
    int Priority; // Do uu tien moi nut
    Data Info;
    Node *pNext;
};
```

Hàm cài đặt thao tác thêm nút mới vào hàng đợi ưu tiên.

```
void EnQueue(Queue *q, Node *p)
{
    if (q->pFront == NULL) // TH1. Hang doi rong
    {
        q->pFront = p;
        q->pRear = p;
    }
    else
    {
        // Tim vi tri phu hop: dau, giua, cuoi hang doi
        if (q->pFront->Priority > p->Priority) // TH2. Them
            dau
        {
            p->pNext = q->pFront;
            q->pFront = p;
        }
        else if (q->pRear->Priority < p->Priority) // TH3.
            Them cuoi
        {
            q->pRear->pNext = p;
            q->pRear = p;
        }
        else // TH4. Them giua
        {
            Node *i = q->pFront;
            while (i->pNext->Priority < p->Priority)
            {
                if (i->pNext == NULL)
                    break;
                i = i->pNext;
            }
```

```
30            p->pNext = i->pNext;
31            i->pNext = p;
32            if (i == q->pRear)
33               q->pRear = p;
34         }
35      }
36      q->Count++;
37   }
```

8.3 Bài tập thực hành

1. Cài đặt hàng đợi sử dụng mảng 1 chiều.
2. Cài đặt hàng đợi phiên bản thêm phần tử vào đầu và lấy ra ở cuối một danh sách liên kết.

Buổi 9

CÂY

9.1 Thông tin chung

Mục tiêu thực hành:

- Định nghĩa cấu trúc dữ liệu của cây tổng quát.
- Các thao tác cơ bản trên cây tổng quát: thêm nút có khóa k, duyệt cây, ...

Công cụ thực hành:

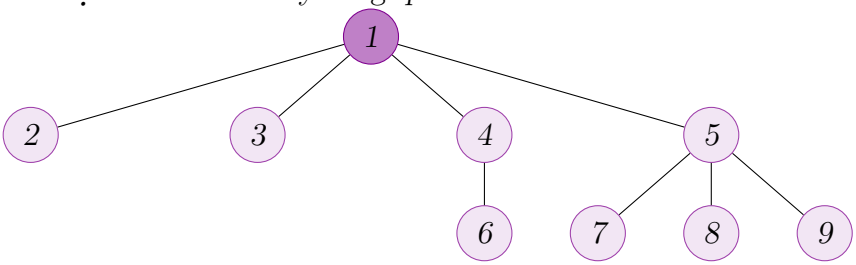
- Ngôn ngữ lập trình: C, C++.
- Môi trường lập trình: Visual C++ 2010 trở lên.

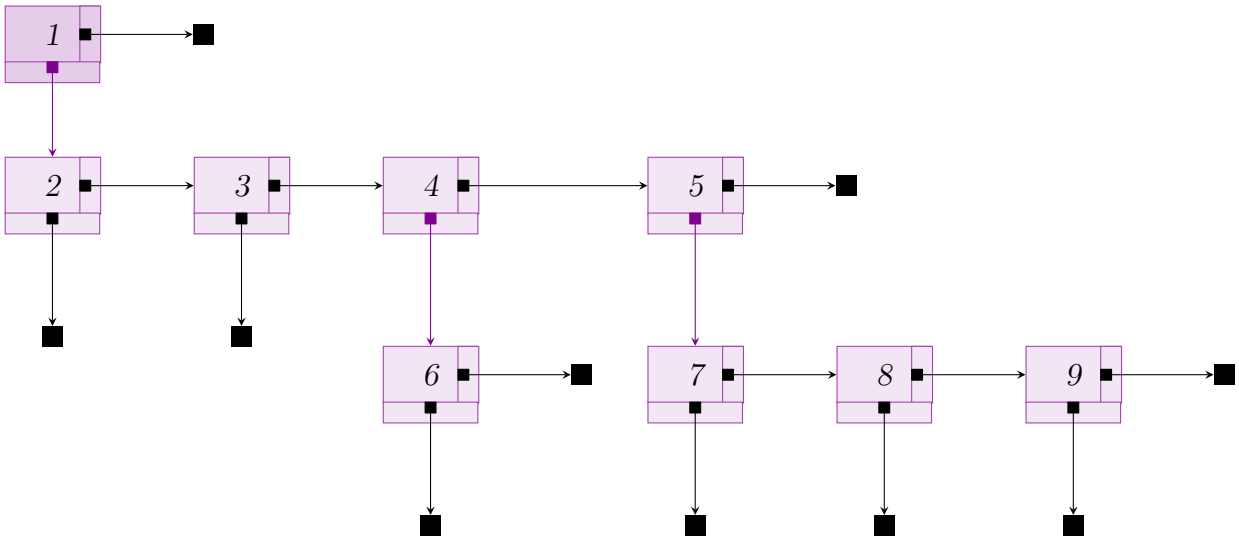
Thời gian thực hành: 3 tiết.

9.2 Nội dung thực hành

9.2.1 Định nghĩa cấu trúc cây tổng quát

Ví dụ 9.2.1. *Cho cây tổng quát như hình sau:*





Ví dụ 9.2.2. *Định nghĩa cấu trúc dữ liệu một nút gồm: thành phần dữ liệu Key và thành phần liên kết pFirstChild (nút con của nút đang xét), pNextSibling (nút anh em của nút đang xét)*

```
typedef int Data;
struct Node
{
    Data Key;
    Node *pFirstChild;
    Node *pNextSibling;
};

void InitTree(Node *&pRoot)
{
    pRoot = NULL;
}
```

9.2.2 Thao tác thêm nút

Ví dụ 9.2.3. *Thêm một nút có khóa k là anh em của nút q cho trước*

```
Node *InsertSibling(Node *q, Data k)
{
    Node *p = InitNode(k);
    if (q == NULL)
        return NULL;
    // Nut anh em cuoi cung cua nut q
    while (q->pNextSibling != NULL)
        q = q->pNextSibling;
    return q->pNextSibling = p;
```

```
}
```

Ví dụ 9.2.4. *Thêm một nút có khóa k vào cây tổng quát*

```
Node *InsertChild(Node *&pRoot, Data k)
{
    Node *p = InitNode(k);

    if (pRoot == NULL) // TH1. Cay rong
        return pRoot = p;
     // TH2. Cay khac rong
    if (pRoot->pFirstChild != NULL)
        return InsertSibling(pRoot->pFirstChild, k);
    else
        return pRoot->pFirstChild = p;
}
```

9.2.3 Thao tác duyệt cây

Ví dụ 9.2.5. *Hàm Traverse() duyệt cây tổng quát.*

```
void Traverse(Node *pRoot)
{
    if (pRoot == NULL)
        return;

    while (pRoot != NULL)
    {
        cout << pRoot->Key << " ";

        // Duyet cay con truoc
        if (pRoot->pFirstChild)
            Traverse(pRoot->pFirstChild);
        // Duyet nut anh em cua nut dang xet
        pRoot = pRoot->pNextSibling;
    }
}
```

Ví dụ 9.2.6. *Áp dụng hàm duyệt cây đếm tổng số nút, tính tổng giá trị các nút trong cây tổng quát.*

```
void Count(Node *pRoot, int &count)
{
    if (pRoot == NULL)
```

```
4       return;
5    while ( pRoot != NULL)
6    {
7       count++;
8       if (pRoot->pFirstChild != NULL)
9          Count(pRoot->pFirstChild, count);
10      pRoot = pRoot->pNextSibling;
11   }
12}
13void Sum(Node *pRoot, int &sum)
14{
15   if (pRoot == NULL)
16      return;
17   while ( pRoot != NULL)
18   {
19      sum += pRoot->Key;
20      if (pRoot->pFirstChild != NULL)
21         Sum(pRoot->pFirstChild, sum);
22      pRoot = pRoot->pNextSibling;
23   }
24}
```

9.3 Bài tập thực hành

Cài đặt hàm thực hiện thao tác:

1. Tìm nút có khóa k.
2. Tìm nút có giá trị lớn nhất trong cây.
3. Đếm số nút lá trong cây.

Buổi 10

CÂY NHỊ PHÂN TÌM KIẾM

10.1 Thông tin chung

Mục tiêu thực hành:

- Định nghĩa cấu trúc dữ liệu của cây nhị phân tìm kiếm *(Binary Search Tree-BST)*.
- Các thao tác cơ bản trên cây NPTK: 3 phương pháp duyệt cây, thêm nút, xóa nút, tìm kiếm, ...

Công cụ thực hành:

- Ngôn ngữ lập trình: C, C++.
- Môi trường lập trình: Visual C++ 2010 trở lên.

Thời gian thực hành: 3 tiết.

10.2 Nội dung thực hành

10.2.1 Định nghĩa cấu trúc cây NPTK

Ví dụ 10.2.1. *Định nghĩa cấu trúc dữ liệu của một nút trong cây NPTK.*

```
1  typedef int Data;
2  struct Node
3  {
4     Data Key;
5     Node *pLeft;
6     Node *pRight;
7  };
8
9  void InitTree(Node *&pRoot)
10 {
11    pRoot = NULL;
12 }
```

10.2.2 Thao tác thêm nút

Ví dụ 10.2.2. *Thêm một nút có khóa k dựa vào tính chất của cây NPTK: cây con bên trái nhỏ hơn nút gốc đang xét và cây con bên phải lớn hơn nút gốc đang xét.*

```
void InsertNode(Node *&pRoot, Data k)
{
    // TH 1. Cay rong
    if (pRoot == NULL)
    {
        Node* p = new Node;
        p->Key = k;
        p->pLeft = NULL;
        p->pRight = NULL;
        pRoot = p;
    }
    else
    {
        if (pRoot->Key > k)
            InsertNode(pRoot->pLeft, k);
        else
            InsertNode(pRoot->pRight, k);
    }
}
```

10.2.3 Thao tác duyệt cây

Ví dụ 10.2.3. *Hàm cài đặt phương pháp duyệt tiền thứ tự (NLR) đối với cây NPTK.*

```
void NLR(Node *pRoot)
{
    if (pRoot != NULL)
    {
        cout << pRoot->Key << "->";
        NLR(pRoot->pLeft);
        NLR(pRoot->pRight);
    }
}

void LNR(Node *pRoot)
{
    // ...
}

```

```
16 void LRN(Node *pRoot)
17 {
18     // ...
19 }
```

Ví dụ 10.2.4. *Hàm cài đặt thao tác tìm nút có khóa k. Trong hàm có 2 lời gọi đệ quy tương ứng với cây con bên trái và cây con bên phải nút đang xét.*

```
1  Node * SearchNode(Node *pRoot, Data k)
2  {
3      if (pRoot != NULL)
4      {
5          if (pRoot->Key > k)  // TH1
6              SearchNode(pRoot->pLeft, k);
7          else if (pRoot->Key < k) // TH2
8              SearchNode(pRoot->pRight, k);
9          else // TH3
10             return pRoot;
11     }
12     return NULL;
13 }
```

10.2.4 Thao tác xóa nút có khóa k

Ví dụ 10.2.5. *Hàm cài đặt thao tác xóa nút có khóa k.*

- *Tìm nút thế mạng của cây đang xét. Chọn nút thế mạng là nút phải nhất/lớn nhất của cây.*

```
1  void SearchStandFor(Node *&pRoot, Node *&p)
2  {
3      // Duyet theo cay con ben phai
4      if (pRoot->pRight != NULL)
5          SearchStandFor(pRoot->pRight, p);
6      else
7      {
8          p->Key = pRoot->Key;
9          p = pRoot;
10         pRoot = pRoot->pLeft;
11     }
12 }
```

- *Hàm cài đặt thao tác xóa một nút có khóa k trong cây NPTK.*

```
1  void RemoveNode(Node *&pRoot, Data k)
2  {
3     Node *p;
4     if (pRoot == NULL)
5     {
6        cout << "Khong tim thay nut co khoa k";
7        return;
8     }
9
10    if (pRoot->Key > k)  // TH 1.
11       RemoveNode(pRoot->pLeft, k);
12    else if (pRoot->Key < k)   // TH 2.
13       RemoveNode(pRoot->pRight, k);
14    else // TH 3.
15    {
16       p = pRoot;
17
18       if (p->pRight == NULL)
19          pRoot = p->pLeft;
20       else if (p->pLeft == NULL)
21          pRoot = p->pRight;
22       else
23          SearchStandFor(pRoot->pLeft, p);
24       delete p;
25    }
26 }
```

10.3 Bài tập thực hành

Cài đặt hàm thực hiện thao tác:

1. Tìm nút có giá trị lớn nhất, nút có giá trị nhỏ nhất trong cây NPTK.
2. Cài đặt lại hàm tìm phần tử thay thế/thế mạng trong trường hợp chọn phần tử trái nhất/nhỏ nhất của cây.