SCC-210 – Algoritmos Avançados

STL: Filas de Prioridade, Conjuntos e Ordenação

João Batista

Fila de Prioridade

- É uma fila na qual os elementos de maior prioridade são removidos primeiro.
- Na implementação em STL, os elementos de maior prioridade são aqueles de maior valor.
- Filas de prioridade não possuem iteradores. Portanto, não se pode percorrer a fila e processar os elementos.

Fila de Prioridade

- Características:
  - Inserção e remoção em tempo logarítmico;
  - Retorno do elemento de maior prioridade em tempo constante.

- Portanto, tem-se uma implementação por *heaps*.

Fila de Prioridade

- São úteis, por exemplo, para:
  - Implementar algoritmo de Dijkstra;
  - Implementar Prim e Kruskal;
  - Implementar buscas heurísticas como A*;
  - Implementar métodos de ordenação.

Fila de Prioridade

- Filas de prioridade
  - #include <queue>
  - Declaração: priority_queue <T>
  - Operações:
    - q.size() – Fornece o número de elementos na fila de prioridade;
    - q.empty() – Fornece true se a fila de prioridade estiver vazia;
    - q.push(E) – Insere E no final da fila;
    - q.top() – Retona o elemento no topo da fila de prioridade (i.e. o elemento de maior valor);
    - q.pop() – Remove o elemento no topo da fila de prioridade;
    - q1 == q2 – true se q1 e q2 possuem os mesmos elementos.

Exemplo: Fila de Prioridade

```
#include<cstdio>
#include<queue>

using namespace std;

main() {
    priority_queue<int> pq;

    pq.push(5);
    pq.push(1);
    pq.push(3);

    printf("%d\n", pq.top());
    pq.pop();
    printf("%d\n", pq.top());
    pq.pop();
    printf("%d\n", pq.top());
    pq.pop();

    if (pq.empty())
        printf("Fila vazia\n");
}
```

Exemplo: Fila de Prioridade

```
#include<cstdio>
#include<queue>

using namespace std;

main() {
    priority_queue<int, vector<int>, greater<int> > pq;

    pq.push(5);
    pq.push(1);
    pq.push(3);

    printf("%d\n", pq.top());
    pq.pop();
    printf("%d\n", pq.top());
    pq.pop();
    printf("%d\n", pq.top());
    pq.pop();

    if (pq.empty())
        printf("Fila vazia\n");
}
```

Exemplo: Fila de Prioridade

```
#include<cstdio>
#include<queue>

using namespace std;

struct aresta {
    int fonte, destino, peso;

    bool operator<(const aresta &v) const {
        return peso < v.peso;
    }
};

int main() {
    priority_queue<aresta> pq;
    aresta v, u;

    v.fonte = 1; v.destino = 2; v.peso = 10;
    pq.push(v);
    v.fonte = 3; v.destino = 5; v.peso = 5;
    pq.push(v);

    u = pq.top();
    printf("%d\n", u.peso);
}
```

Conjuntos

- São containers ordenados que permitem o uso simples de algoritmos de união, intersecção, e cálculo de diferenças.
- Por serem conjuntos ordenados, é importante definir uma relação de ordem (<).
- Existem duas classes containers:
  - Set: conjunto cujas chaves não podem se repetir;
  - Multiset: “conjunto” cujas chaves podem se repetir.

Sets

- #include <set>
- Declaração: set <T>
- Operações:
  - s.insert(E) – Insere E no conjunto, se E já pertence ao conjunto, nada é feito;
  - s.erase(E) – Remove E do conjunto, se E não pertence ao conjunto, nada é feito;
  - s.find(E) – Localiza E no conjunto. Se E não pertence ao conjunto, retorna s.end();
  - s.size() – Fornece o número de elementos do conjunto;
  - s.empty() – Retorna true se o conjunto for vazio;
  - s.clear() – Apaga todos os elementos do conjunto;
  - s1 == s2 – true se s1 e s2 possuírem os mesmos elementos.

Sets

- Continuação:
  - s.count(E) – Retorna o número de elementos cuja chave é igual a E (resposta 0 ou 1);
  - s.lower_bound(E) – Retorna o primeiro elemento cuja chave é não menor do que E;
  - s.upper_bound(E) – Retorna o primeiro elemento cuja chave é maior do que E;

Sets - Exemplo

```
#include<set>
#include<cstdio>;

using namespace std;

int main() {
    set<int> a;
    set<int>::iterator i;

    a.insert(1); a.insert(3);
    a.insert(5); a.insert(3);

    for (i=a.begin(); i!=a.end(); i++)
        printf("%d ", *i);              // 1 3 5 ordenado
}
```

Multisets

- #include <set>
- Declaração: multiset <T>
- Operações:
  - s.insert(E) – Insere E no conjunto, se E já pertence ao conjunto, passa a existir uma outra cópia de E;
  - s.erase(E) – Remove E do conjunto, se E não pertence ao conjunto, nada é feito;
  - s.find(E) – Localiza E no conjunto. Se E não pertence ao conjunto, retorna s.end();
  - s.size() – Fornece o número de elementos do conjunto;
  - s.empty() – Retorna true se o conjunto for vazio;
  - s.clear() – Apaga todos os elementos do conjunto;
  - s1 == s2 – true se s1 e s2 possuírem os mesmos elementos.

Multisets

- Continuação:
  - s.count(E) – Retorna o número de elementos cuja chave é igual a E;
  - s.lower_bound(E) – Retorna o primeiro elemento cuja chave é não menor do que E;
  - s.upper_bound(E) – Retorna o primeiro elemento cuja chave é maior do que E;

Multisets - Exemplo

```
#include<set>
#include<cstdio>;

using namespace std;

int main() {
    multiset<int> a;
    multiset<int>::iterator i;

    a.insert(1); a.insert(3);
    a.insert(5); a.insert(3);

    for (i=a.begin(); i!=a.end(); i++)
        printf("%d ", *i);              // 1 3 3 5 ordenado
}
```

Algoritmos sobre conjuntos

- Tanto para set quanto para multiset, STL provê alguns algoritmos clássicos de manipulação.
- Esses algoritmos não são métodos dessas classes, mas funções declaradas no arquivo de inclusão: algorithm (#include<algorithm>).

Algoritmos sobre conjuntos

- As principais funções são:
  - set_union – Realiza a união entre dois conjuntos:
    - Set: união clássica entre conjuntos;
    - Multiset: no caso de elementos repetidos max(m,n);
    - Complexidade linear.
  - set_intersection – Realiza a interseção entre conjuntos:
    - Set: intersecção clássica entre conjuntos;
    - Multiset: no caso de elementos repetidos min(m,n);
    - Complexidade linear.

Algoritmos sobre conjuntos

- Continuação:
  - set_difference – Realiza a diferença entre dois conjuntos:
    - Set: diferença clássica entre conjuntos:
      - A – B = elementos em A que não ocorrem em B.
    - Multiset: no caso de elementos repetidos max(m-n, 0);
    - Complexidade linear.

Multisets - Exemplo

```
#include<algorithm>
#include<set>
#include<cstdio>;

using namespace std;

int main() {
    set<int> a, b, c, d, e;

    a.insert(1); a.insert(3); a.insert(5);
    b.insert(2); b.insert(3); b.insert(6);

    set_union(a.begin(), a.end(), b.begin(), b.end(),
              inserter(c, c.begin()));

    set_intersection(a.begin(), a.end(), b.begin(), b.end(),
              inserter(d, d.begin()));

    set_difference(a.begin(), a.end(), b.begin(), b.end(),
              inserter(e, e.begin()));
}
```

Conjuntos como Arranjo de Bits

- Arranjo binário de n posições $\rightarrow$ subconjuntos de n elementos:
  - Bit $i = 1 \rightarrow$ i-ésimo elemento pertence ao subconjunto.
  - Bit $i = 0 \rightarrow$ caso contrário.
- Representação mais simples e conveniente para subconjuntos derivados de universos estáticos e de tamanho moderado.
- Inserção e remoção por chaveamento de bit.
- Interseção e união via operações lógicas AND e OR nos arranjos.
- Eficiente em termos de memória:
  - Arranjo de 1000 inteiros de 4 bytes pode representar qualquer subconjunto de 32000 elementos!

Conjuntos como Arranjo de Bits

- Nesse caso precisamos de vetores de bits, que podem ser implementados:
  - vector<bool>
    - É um container STL e possui iteradores e demais recursos.
  - bitset<N>
    - Não é um container STL, e portanto não possui iteradores
    - Tem tamanho fixo especificado por N

Ordenação em C

- C provê a função qsort que implementa o algoritmo QuickSort (stdlib.h)

```c
#include <stdio.h>
#include <stdlib.h>

int values[] = { 40, 10, 100, 90, 20, 25 };

int compare (const void * a, const void * b) {
  return ( *(int*)a - *(int*)b );
}

int main () {
  int n;
  qsort (values, 6, sizeof(int), compare);
  for (n=0; n<6; n++)
     printf ("%d ",values[n]);
  return 0;
}
```

http://www.cplusplus.com

Ordenação em C

- C provê a função bsearch que implementa a busca binária (stdlib.h)

```c
#include <stdio.h>
#include <stdlib.h>

int compareints (const void * a, const void * b) {
  return ( *(int*)a - *(int*)b );
}

int values[] = { 10, 20, 25, 40, 90, 100 };

int main () {
  int * pItem;
  int key = 40;
  pItem = (int*) bsearch (&key, values, 6, sizeof (int), compareints);
  if (pItem!=NULL)
    printf ("%d is in the array.\n",*pItem);
  else
    printf ("%d is not in the array.\n",key);
  return 0;
}
```

http://www.cplusplus.com

Ordenação em STL

- C++ standard library provê mais de um algoritmo de ordenação. Alguns dos principais são:
  - **sort()** – geralmente baseado no quicksort: $O(n \log n)$ na média e $O(n^2)$ no pior caso;
  - **partial_sort()** – geralmente baseado no heapsort. Tem pior caso $O(n \log n)$, mas para a maior parte das entradas é de 2 a 5 vezes mais lento que o quicksort. Pode parar a ordenação quando os k primeiros elementos estão ordenados.
  - **stable_sort()** – geralmente baseado no mergesort. Preserva a ordem os elementos de mesmo valor (estável). Pode ter complexidade $O(n \log n)$ ou $O(n \log n \log n)$.

Ordenação em STL

- Os algoritmos de ordenação sort e stable_sort possuem as seguintes formas de chamada:
  - void sort(RandomAccessIterator beg, RandomAccessIterator end)
  - void sort(RandomAccessIterator beg, RandomAccessIterator end, BinaryPredicate op)
- Ordenam os elementos no intervalo [beg, end) com o operador <.
- op é um operador que pode substituir <.
- É necessário que o container tenha suporte a iteradores aleatórios (ex. vectors e deques).

Exemplo sort 1

```
#include<cstdio>
#include<vector>
#include<algorithm>

using namespace std;

main()
{
    vector<int> v;

    v.push_back(5); v.push_back(2);
    v.push_back(1); v.push_back(7);
    v.push_back(4); v.push_back(8);

    sort(v.begin(), v.end());

    for (int i = 0; i < v.size(); i++)
        printf("%d ", v[i]);
    return 0;
}
```

Exemplo sort 2

```
#include<cstdio>
#include<vector>
#include<algorithm>

using namespace std;

bool myfunction (int i,int j) { return (j<i); }

main()
{
    vector<int> v;

    v.push_back(5); v.push_back(2);
    v.push_back(1); v.push_back(7);
    v.push_back(4); v.push_back(8);

    sort(v.begin(), v.end(), myfunction);

    for (int i = 0; i < v.size(); i++)
        printf("%d ", v[i]);
    return 0;
}
```

Exemplo sort 3

```
#include<cstdio>
#include<vector>
#include<algorithm>
#include<functional>    // Para acessar greater

using namespace std;

main()
{
    vector<int> v;

    v.push_back(5); v.push_back(2);
    v.push_back(1); v.push_back(7);
    v.push_back(4); v.push_back(8);

    sort(v.begin(), v.end(), greater<int>());

    for (int i = 0; i < v.size(); i++)
        printf("%d ", v[i]);
    return 0;
}
```

Ordenação estável: stable_sort

- Em ordenação estável, a ordem relativa das chaves de mesmo valor é mantida pela ordenação.
  - Útil quando precisa ordenar por múltiplas chaves

Chave principal ⇨ A B A B C B D
Chave secundária ⇨ 1 2 3 4 5 6 7

⇩ Ordenação estável

Chave principal ⇨ A A B B B C D
Chave secundária ⇨ 1 3 2 4 6 5 7

Ordenação em STL

- O algoritmo de ordenação partial_sort pode interromper a ordenação quando os primeiros elementos (até sortEnd) estão ordenados.
  - void partial_sort(RandomAccessIterator beg,
    RandomAccessIterator sortEnd,
    RandomAccessIterator end)
  - void partial_sort(RandomAccessIterator beg,
    RandomAccessIterator sortEnd,
    RandomAccessIterator end,
    BinaryPredicate op)
- Útil quando se tem interesse em apenas os *k* valores maiores (ou menores) de uma sequência.