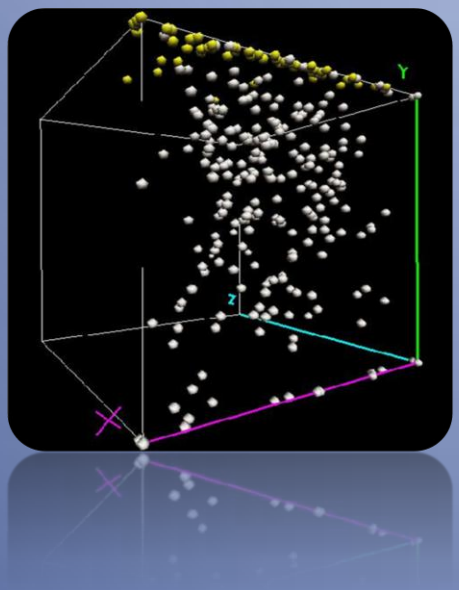

ALGORITMOS E ESTRUTURAS DE DADOS III

Tutorial 13 (usa o compilador de linguagem C Dev-C++ versão 4.9.9.2)

Parte 1 de 2 sobre ordenação externa: seleção por substituição.

1 INTRODUÇÃO

Esta série de tutoriais sobre *Algoritmos e Estruturas de Dados III* foi escrita usando o **Microsoft Windows 7 Ultimate**, **Microsoft Office 2010**, **Bloodshed Dev-C++** versão 4.9.9.2 (pode ser baixado em http://www.bloodshed.net), **Code::Blocks** versão 10.05 (pode ser baixado em http://www.codeblocks.org) referências na internet e notas de aula do professor quando estudante. Ela cobre desde os algoritmos de ordenação, passando pela pesquisa em memória primária e culminando com a pesquisa em memória secundária.

Nós entendemos que você já conhece o compilador Dev-C++. No caso de você ainda não o conhecer, dê uma olhada nos tutoriais Dev-C++ 001 a 017, começando pelo Tutorial Dev-C++ - 001 - Introdução.

Se não tem problemas com a linguagem C/C++ e o compilador Dev-C++, então o próximo passo é saber ler, criar e alterar arquivos em disco usando linguagem C/C++. Se ainda não sabe como fazê-lo, dê uma olhada nos tutoriais Dev-C++ 001 e 002, começando pelo Tutorial Dev-C++ 001 – Criação, Leitura e Alteração de Arquivos.

Se sabe todas as coisas anteriores, então a próxima etapa é conhecer os algoritmos mais básicos de ordenação. Em minhas notas de aula você encontra um material básico, porém detalhado e com algoritmos resolvidos, dos principais métodos de ordenação existentes.

Adotaremos o livro ***Projeto de Algoritmos com Implementação em Pascal e C***, Editora Cengage Learning, de Nivio Ziviani, como livro-texto da disciplina. Nele você encontrará os métodos de ordenação que iremos estudar.

Seu próximo passo será estudar os algoritmos de ordenação por **Inserção**, **Seleção**, **Shellsort**, **Heapsort** e **Quicksort**. Você pode usar os links anteriores ou fazer uso do livro-texto.

Se você estiver lendo este tutorial tenha certeza de ter seguido os Tutoriais AED 001 a 012. Agora que você seguiu todos os passos até aqui, está pronto para prosseguir com este tutorial.

2 IMPLEMENTAÇÃO

A implementação do método de intercalação balanceada pode ser realizada utilizando-se filas de **prioridades**. Tanto a passada inicial para quebrar o arquivo em blocos ordenados quanto a fase de intercalação podem ser implementadas de forma eficiente e elegante utilizando-se filas de prioridades. A operação básica necessária para formar os blocos ordenados iniciais corresponde a obter o menor dentre os registros presentes na memória interna, o qual deve ser substituído pelo próximo registro da fita de entrada. A operação de substituição do menor item de uma fila de prioridades implementada através de um *heap* é a operação ideal para resolver o problema.

A operação de substituição corresponde a retirar o menor item da fila de prioridades, colocando no seu lugar um novo item, seguido da reconstituição da propriedade do *heap*.

Para cumprir esta primeira passada nós iniciamos o processo fazendo *m* inserções na fila de prioridades inicialmente vazia. A seguir o menor item da fila de prioridades é substituído pelo próximo item do arquivo de entrada, com o seguinte passo adicional: se o próximo item é menor que o que está saindo (o que significa que este item não pode fazer parte do bloco ordenado corrente), então ele deve ser marcado como membro do próximo bloco e tratado como maior do que todos os itens do bloco corrente. Quando um item marcado vai para o **topo da fila de prioridades**, o bloco corrente é encerrado e um novo bloco ordenado é iniciado. A FIGURA 1 mostra o resultado da primeira passada sobre o arquivo da FIGURA 4. Os asteriscos indicam quais chaves na fila de prioridades pertencem a blocos diferentes.

Entra	1	2	3
E	I	N	T
R	N	E*	T
C	R	E*	T
A	T	E*	C*
L	A*	E*	C*
A	C*	E*	L*
C	E*	A	L*
A	L*	A	C
O	A	A	C
B	A	O	C
A	B	O	C
L	C	O	A*
A	L	O	A*
N	O	A*	A*
C	A*	N*	A*
E	A*	N*	C*
A	C*	N*	E*
D	E*	N*	A
A	N*	D	A
	A	D	A
	A	D	
	D		

FIGURA 1: RESULTADO DA PRIMEIRA PASSADA USANDO SELEÇÃO POR SUBSTITUIÇÃO

Cada linha da FIGURA 1 representa o conteúdo de um *heap* de tamanho três. A condição do *heap* é que a primeira chave tem que ser menor do que a

segunda e a terceira chaves. Nós iniciamos com as três primeiras chaves do arquivo, as quais já formam um *heap*. A seguir, o registro *I* sai e é substituído pelo registro E, que é menor do que a chave *I*. Neste caso, o registro *E* não pode ser incluído no bloco corrente: ele é marcado e considerado maior do que os outros registros do *heap*. Isto viola a condição do *heap*, e o registro *E** é trocado com o registro *N* para reconstituir o *heap*. A seguir, o registro *N* sai e é substituído pelo registro *R*, o que não viola a condição do heap. Ao final do processo vários blocos ordenados são obtidos. Esta forma de utilizar filas de prioridades é chamada seleção por substituição (vide Knuth, 1973, Seção 5.4.1; Sedgewick, 1988, p.180).

Para uma memória interna capaz de reter apenas 3 registros é possível produzir os blocos ordenados INRT, ACEL, AABCLO, AACEN e AAD, de tamanhos 4, 4, 6, 5 e 3, respectivamente. Knuth (1973, pp. 254-256) mostra que, se as chaves são randômicas, os blocos ordenados produzidos são cerca de duas vezes o tamanho dos blocos produzidos por ordenação interna. Assim, a fase de intercalação inicia com blocos ordenados em média duas vezes maiores do que o tamanho da memória interna, o que pode salvar uma passada na fase de intercalação. Se houver alguma ordem nas chaves então os blocos ordenados podem ser ainda maiores. Ainda mais, se nenhuma chave possui mais do que m chaves maiores do que ela, antes dela, então o arquivo é ordenado já na primeira passada. Por exemplo, o conjunto de registros RAPAZ é ordenado pela primeira passada, conforme ilustra a FIGURA 2.

Entra	1	2	3
A	A	R	P
Z	A	R	P
	P	R	Z
	R	Z	
	Z		

FIGURA 2: CONJUNTO ORDENADO NA PRIMEIRA PASSADA

A fase de intercalação dos blocos ordenados obtidos na primeira fase também pode ser implementada utilizando-se uma fila de prioridades. A operação básica para fazer a *intercalação-de-f-caminhos* é obter o menor item dentre os itens ainda não retirados dos *f* blocos a serem intercalados.

Para tal basta montar uma fila de prioridades de tamanho *f* a partir de cada uma das *f* entradas. Repetidamente, substitua o item no topo da fila de prioridades (no caso o menor item) pelo próximo item do mesmo bloco do item que está sendo substituído, e imprima em outra fita o elemento substituído. A FIGURA 3 mostra o resultado da intercalação de INT com CER com AAL, os quais correspondem aos blocos iniciais das fitas 1, 2 e 3 mostrados na FIGURA 5.

Quando *f* não é muito grande não há vantagem em utilizar seleção por substituição para intercalar os blocos, pois é possível obter o menor item

Entra	1	2	3
A	A	C	I
L	A	C	I
E	C	L	I
R	E	L	I
N	I	N	R
	L	R	R
T	N	T	
	R		
	T		

FIGURA 3: INTERCALAÇÃO USANDO SELEÇÃO POR SUBSTITUIÇÃO

fazendo $f - 1$ comparações. Quando *f* é 8 ou mais, é possível ganhar tempo usando um *heap* como mostrado acima. Neste caso cerca de $\log_2 f$ comparações são necessárias para obter o menor item.

3 CONSIDERAÇÕES PRÁTICAS

Para implementar o método de ordenação externa descrito anteriormente é muito importante implementar de forma eficiente as operações de entrada saída de dados. Estas operações compreendem a transferência dos dados entre a memória interna e as unidades externas onde estão armazenados os registros a serem ordenados. Deve-se procurar realizar a leitura, a escrita o processamento interno dos dados de forma simultânea. Os computadores de maior porte possuem uma ou mais unidades independentes para processamento de entrada e saída que permitem realizar simultaneamente as operações de entrada, saída e processamento interno.

Knuth (1973) discute várias técnicas para obter superposição de entrada saída com processamento interno. Uma técnica comum é a de utilizar $2f$ áreas de entrada e $2f$ áreas de saída. Para cada unidade de entrada ou saída são mantidas duas áreas de armazenamento: uma para uso do processador central e outra para uso do processador de entrada ou saída. Para entrada, o processador central usa uma das duas áreas enquanto a unidade de entrada está preenchendo a outra área. No momento que o

processador central termina a leitura de uma área, ele espera que a unidade de entrada acabe de preencher a outra área e então passa a ler dela, enquanto a unidade de entrada passa a preencher a outra. Para saída, a mesma técnica é utilizada.

Existem dois problemas relacionados com a técnica de utilização de duas áreas de. armazenamento. Primeiro, apenas metade da memória disponível é utilizada, o que pode levar a uma ineficiência se o número de áreas for grande, como no caso de uma *intercalação-de-f-caminhos* para f grande. Segundo, em uma *intercalação-de-f-caminhos* existem f áreas correntes de entrada; se todas as áreas se tornarem vazias aproximadamente ao mesmo tempo, muita leitura será necessária antes de podermos continuar o processamento, a não ser que haja uma previsão de que esta eventualidade possa ocorrer.

Os dois problemas podem ser resolvidos com a utilização de uma técnica chamada previsão, a qual requer a utilização de apenas uma área extra de armazenamento (e não f áreas) durante o processo de intercalação. A melhor forma de superpor a entrada com processamento interno durante o processo de seleção por substituição é superpor a entrada da próxima área que precisa ser preenchida a seguir com a parte de processamento interno do algoritmo. Felizmente, é fácil saber qual área ficará vazia primeiro, simplesmente olhando para o último registro de cada área. A área cujo último registro é o menor, será a primeira a se esvaziar; assim nós sempre sabemos qual conjunto de registros deve ser o próximo a ser transferido para a área.

FIGURA 4: ARQUIVO EXEMPLO COM 22 REGISTROS

Por exemplo, na intercalação de INT com CER com AAL nós sabemos que a terceira área será a primeira a se esvaziar.

FIGURA 5: FORMAÇÃO DOS BLOCOS ORDENADOS INICIAIS

Uma forma simples de superpor processamento com entrada na intercalação de vários caminhos é manter uma área extra de armazenamento, a qual é preenchida de acordo com a regra descrita acima. Enquanto os blocos INT, CER e AAL da FIGURA 5 estão sendo intercalados o processador de entrada está preenchendo a área extra com o bloco ACN. Quando o processador central encontrar uma área vazia, ele espera até que a área de entrada seja preenchida caso isto ainda não tenha ocorrido, e então aciona o processador de entrada para começar a preencher a área vazia com o próximo bloco, no caso ABL.

Outra consideração prática importante está na escolha do valor de f, que é a ordem da intercalação. No caso de fita magnética a escolha do valor de f deve ser igual ao número de unidades de fita disponíveis menos um. A fase de intercalação usa f fitas de entrada e uma fita de saída. O número de fitas de entrada deve ser no mínimo dois, pois não faz sentido fazer intercalação com menos de duas fitas de entrada.

No caso de disco magnético o mesmo raciocínio acima é válido. Apesar do disco magnético permitir acesso direto a posições arbitrárias do arquivo, o acesso sequencial é mais eficiente. Logo, o valor de f deve ser igual ao número de unidades de disco disponíveis menos um, para evitar o maior custo envolvido se dois arquivos diferentes estiverem em um mesmo disco.

Sedegwick (1983) apresenta outra alternativa: considerar f grande o suficiente para completar a ordenação em um número pequeno de passadas.

Uma intercalação de duas passadas em geral pode ser realizada com um número razoável para f. A primeira passada no arquivo utilizando seleção por substituição produz cerca de $n/2m$ blocos ordenados. Na fase de intercalação cada etapa divide o número de passadas por f. Logo, f deve ser escolhido tal que

$$f^2 > \frac{n}{2m}.$$

Para n igual a 200 milhões e m igual a 1 milhão então f = 11 é suficiente para garantir a ordenação em duas passadas. Entretanto, a melhor escolha para f entre estas duas alternativas é muito dependente de vários parâmetros relacionados com o sistema de computação disponível.

4 SELEÇÃO POR SUBSTITUIÇÃO

4.1 VISÃO GERAL

Executa a ordenação interna, passando os registros através de uma grande fila de prioridade.

4.2 Estratégia Detalhada

1. Escolha um fila de prioridade tão grande quanto possível, digamos de M elementos.
2. Passo de Ordenação
   a. **Passo de Inicialização**: Leia *M* registros para a fila de prioridade.
   b. **Passo de Substituição** (criando uma passada única):
      i. Remova o menor registro de todos da fila de prioridade e grave-o na saída.
      ii. Leia um registro do arquivo de entrada. Se o novo elemento é menor que o último gravado, ele não pode fazer parte da passada atual. Marque-o como pertencente à próxima passada e trate-o como maior que todos os elementos não marcados na fila.
      iii. Termine a passada quando um elemento marcado chegue ao topo da fila.
3. **Passo de Mesclagem**: Mesmo que o anterior.

4.3 Pseudocódigo

O pseudocódigo abaixo também mostra como o código **create_initial_runs** deve ser modificado do esquema de ordenação por intercalação balanceada por múltiplos caminhos. Os outros procedimentos do esquema de ordenação por intercalação balanceada por múltiplos caminhos permanecem os mesmos.

4.3.1 Código Para Criar Passada Inicial

Listagem 1:

```
create_initial_runs(input_file_name, run_size, num_ways) {
  // allocate a dynamic array, a, large enough
  // to accommodate runs of size run_size
  open the input file using the fields package
  for i = 0 to NUM_WAYS-1 {
     open output_scratch_file i using fopen
  }
  end_of_input = false
  next_output_file = 0
  num_runs_per_output_file = 0

 // instantiate the heap
 for i = 1 to run_size {
   read a record into a[i]
   if (end of input)
      end_of_input = true
      break

   // indicate the record belongs in the current run
  // and insert the record into the heap
   a[i]->mark = current_mark;
   upheap(i);
 }
 // initialize the heap size
 N = i - 1;

 // create the initial runs--get_item sets end_of_input to
 // true only when there is no more input and the heap
 // is exhausted

 while (end_of_input == false) {
   // keep outputting records to the current run
   // until get_item returns a NULL record
   for(record=get_item(input_file, &end_of_input);
        record!=NULL);
       record = get_item(input_file, &end_of_input)) {
       write record to output_scratch_files[next_output_file]
   }
   output the sentinel value to scratch_output_file[next_output_file]

   // everytime we get back to the first output file, increment
   // the number of runs per output file by 1
   if (next_output_file == 0)
      num_runs_per_output_file=num_runs_per_output_file+1
    next_output_file = (next_output_file + 1) % num_ways
 }

 // make sure the same number of runs are assigned to
 // each scratch output file
 if (next_output_file != 0) {
    for i = next_output_file to (num_ways - 1) {
       output the sentinel value to scratch_output_file[i]
    }
 }
 for i = 0 to (num_ways -1)
    close scratch_output_file[i] using fclose
 close the input file using the fields package

 return num_runs_per_output_file
}
```

4.4 Código de Heap

Este código assume que há dois valores em um registro:

- **mark**: Uma marca que alterna entre dois valores arbitrários (e.g., 0 e 1). Este campo é usado para determinar quando um registro pertence à passada atual ou à próxima passada.
- **key**: A chave do registro.

Há também a variável global:

- **current_mark**: Indica em qual dos dois valores uma marca de registro deve ser colocada para que ele pertença à próxima passada.

O código pode ser visto na *listagem 2*, a seguir.

Listagem 2:

```
// if the two records both belong in the run or both
// don't belong in the run, compare the two values and
// return true if the first value is less than the first.
// Otherwise, return true if the first record belongs in
// the run, and false otherwise (in the latter case, the
// second record belongs in the run and hence it is
// "less than" the first record).

int less_than_or_equal(record1, record2) {
  if (record1->mark == record2->mark)
    return (record1->key <= record2->key)
  else
    return (record1->mark == current_mark)
}

downheap (int k) {
   int j;
   Record v;

   v = a[k];
   while (k <= N/2) {
      j = 2 * k;
      if (j < N && less_than_or_equal(a[j+1], a[j])) j++;
      if (less_than_or_equal(v, a[j])) break;
      a[k] = a[j];
      k = j;
   }
   a[k] = v;
}

int heap_empty () {
  return (N == 0);
}
```

```
// get_item returns the next item in the run. It returns
// NULL if the end of a run or end of the input is reached.
// The flag end_of_input allows get_item to indicate whether
// the reason for the NULL is due to the end of a run or
// the end of the input.
// end_of_input must be a pointer to allow a value to
// be passed back through the end_of_input parameter.

PERSONNEL get_item(input_file, *end_of_input) {
  static int more_input = true;

  // determine if the heap is empty. If it is, then the end of the
  // input has been reached, so set end_of_input to true
  if (heap_empty()) {
    *end_of_input = true;
    return NULL;
  }
  // determine if the root element belongs in this run
  if (a[1]->mark != current_mark) {
    // reverse the mark so that all the elements in the heap
    // will be available for the next run
    current_mark = !(current_mark);
    *end_of_input = false;
    return NULL;
  }

  if (more_input) {
    if (not end of file) {
      read a record into a[0]
      // determine if the newly read record belongs in
      // the current run or the next run.
      if (a[0]->key >= a[1]->key)
         a[0]->mark = current_mark
      else
         a[0]->mark = !(current_mark)
      // insert the newly read record into the heap by executing
      // a replace operation (i.e., the root of the heap will be
      // returned and the new value will be pushed onto
      // the heap)
      downheap(0);
      return a[0];
    }
    else {
    // once the input is exhausted, start returning
    // items from the heap
      more_input = false;
      return remove_item();
    }
  }
  else {
    // if the input has been exhausted, remove the top item
    // on the heap and return it
    return remove_item();
  }
}
```

4.5 PERFORMANCE

1. Para chaves aleatórias, as execuções produzidas pela seleção de substituição são cerca de duas vezes o tamanho do *heap* usado.
2. Consequentemente, filas de prioridade economizam aproximadamente uma mesclagem. O número de passagens é aproximadamente $1 + \log P \times \left(\frac{N}{2M}\right)$.

4.6 Considerações Práticas

4.6.1 Problemas de Sistemas

O custo de E/S domina o custo de computação sobrepondo leitura, escrita, e computação críticas. Um arquivo é dividido em blocos, cada um com um número fixo de bytes. Quando um arquivo é lido na memória ou escrito, o sistema mantém buffers que são do tamanho do bloco de um arquivo. Arquivos são transferidos para e da memória em blocos igual ao tamanho de bloco de um arquivo, ou equivalentemente, igual ao tamanho de um *buffer*.

1. **Buffer Duplo**: Para cada arquivo de entrada mantenha dois *buffers* – um que é usado pela CPU e outro que é usado pelo processador de E/S. Enquanto a CPU usa seus *buffers* o processador de E/S lê os seus.
2. **Previsão**: O inconveniente do *Buffer Duplo* é que ele usa apenas metade do espaço de memória disponível. Se houver P unidades de entrada sendo usadas, $2P$ buffers são necessários. *Previsão* pode reduzir esse número para $P + 1$.
   A ideia é a seguinte:
   a. Encontre o *buffer* de entrada que será esgotado em seguida. O *buffer* que vai ser esgotado em seguida é aquele que tem o menor item no final.
   b. Reserve o *buffer* extra para esse arquivo de entrada e comece a preenchê-lo.

4.6.2 Selecionando P

1. Se N fitas zeradas estão disponíveis, use ordenação de $(N/2)$-*caminhos* (i.e. $P = N/2$).
2. Se discos estão disponíveis, uma boa estratégia é escolher P de forma que só 2 passagens sejam necessárias. Isto requer que:

$$\log P \times \left(\frac{N}{2M}\right) < 2$$
$$\Rightarrow \frac{N}{2M} < 2^P$$
$$\Rightarrow \sqrt{\frac{N}{2M}} < P$$

5 Notas Bibliográficas

Knuth (1973) é a referência mais completa que existe sobre ordenação em geral. Outros livros interessantes sobre o assunto são *Sedgewick* (1988), *Wirth* (1976), *Cormem*, *Leiserson* e *Rivest* (1990), *Aho*, *Hoperoft* e *Ullman* (1983). 0 livro de *Gonnet* e *Baeza-Yates* (1991) apresenta uma relação bibliográfica extensa e atualizada sobre o assunto.

Shellsort foi proposto por *Shell* (1959).

Quicksort foi proposto por *Hoare* (1962). Um estudo analítico detalhado, bem como um estudo exaustivo dos efeitos práticos de muitas modificações e melhorias sugeridas para o Quicksort pode ser encontrado em Sedgewick (1975) e *Sedgewick* (1978a).

Heapsort foi proposto por *Williams* (1964) e melhorado por *Floyd* (1964).

6 Exercícios

1. Dado que existe necessidade de ordenar arquivos de tamanhos diversos, podendo também variar o tamanho dos registros de um arquivo para outro, apresente uma discussão sobre quais algoritmos de ordenação você escolheria diante das diversas situações colocadas acima.

   Que observações adicionais você apresentaria caso haja:
   a) restrições de estabilidade; ou

   b) de intolerância para o pior caso (isto é, a aplicação exige um algoritmo eficiente mas não permite que o mesmo eventualmente leve muito tempo para executar).

2. Invente um vetor-exemplo de entrada para demonstrar que *Ordenação por Seleção* é um método instável. Mostre os passos da execução do algoritmo até que a estabilidade é violada. Note que quanto menor for o vetor que você inventar, mais rápido você vai resolver a questão.

3. Considere uma matriz retangular. Ordene em ordem crescente os elementos de cada linha. A seguir, ordene em ordem crescente os elementos de cada coluna. Prove que os elementos de cada linha continuam em ordem.

7 Terminamos

Terminamos por aqui. Mais em http://flavioaf.blogspot.com.

Corra para o próximo tutorial.