Lógica Computacional 117366 - Turma A
Enunciado do Projeto
Formalização de Algoritmos para Ordenação com Heaps

1ro de agosto de 2022 (Semestre 2022-1)

Prof. Mauricio Ayala-Rincón

1 Introdução

Algoritmos de busca e ordenação são fundamentais em Ciência da Computação. Busca é um mecanismo essencial em estruturas de dados e ordenação é relevante para diminuir o tempo de busca em diversas estruturas de dados. Neste projeto considerar-se-ão algoritmos de ordenação sobre o tipo abstrato de dados `finite_sequences` como especificado no assistente de demonstração PVS.

O objetivo do projeto da disciplina é introduzir os mecanismos básicos de manuseio de tecnologias de verificação e formalização que utilizam técnicas dedutivas lógicas, como as estudadas na disciplina, para garantir que objetos computacionais são logicamente corretos [AdM17].

2 Descrição do Projeto

Este projeto aborda questões apresentadas now arquivos de especificação e prova do algoritmo de ordenação por heaps, que são, respectivamente, `heapsort.pvs` e `heapsort.prf`. Adicionalmente, apresenta-se uma questão no arquivo `sorting_seq.pvs`. Os arquivos estão disponíveis na plataforma Aprender3 da disciplina, especificados na linguagem do assistente de demonstração PVS (`pvs.csl.sri.com`) executável em plataformas Unix/Linux e OS. PVS deve ser instalado como extensão de VS code, incluindo a livraria de PVS de NASA. Os alunos deverão formalizar propriedades da especificação para ordenação por `heap` sobre a estrutura de dados de sequências finitas sobre um tipo não interpretado com uma quasiordem.

2.1 Ordenação por heap: *Heapsort*

Algumas funções utilizadas na especificação deste algoritmo estão especificadas para listas no arquivo `sorting.pvs` (arquivo de provas `sorting.prf`), como a noção de `occurrence`, que especifica quantas vezes determinado natural está presente em uma lista, dada por:

```
occurrence(l)(x): RECURSIVE nat =
IF  null?(l) THEN 0
ELSIF
car(l) = x THEN 1 + occurrence(cdr(l))(x)
ELSE
occurrence(cdr(l))(x)
ENDIF
MEASURE length(l)
```

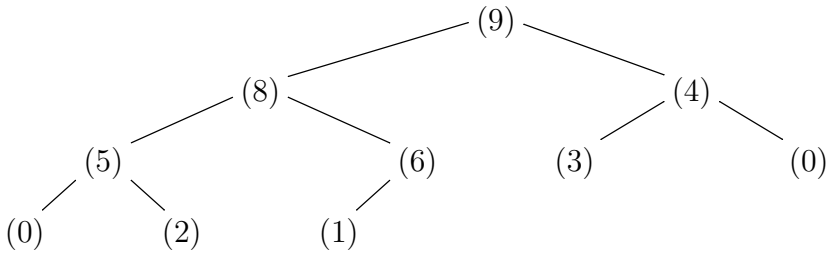
Também temos para sequências finitas uma função semelhante, dada por:

```
occurrences(h)(x:nat): RECURSIVE nat =
IF  length(h) = 0 THEN 0
ELSIF
h(0) = x THEN 1 + occurrences(h^(1,length(h)-1))(x)
ELSE
occurrences(h^(1,length(h)-1))(x)
ENDIF
MEASURE length(h)
```

A teoria heapsort formaliza a correção da especificação de ordenação por heap, abaixo. Ou seja, demonstra que a função heapsort, assim especificada, ordena a entrada preservando o número de ocorrências dos elementos na entrada.

```
heapsort(h) : finite_sequence[nat] =
IF length(h) > 1 THEN
heapsort_aux(heapify(h)(floor(length(h)/2) - 1))(length(h)-1)
ELSE h
ENDIF
```

A função `heapsort` inicialmente transforma a sequência dada em um *heap*, usando a função `heapify`. Um *heap* é uma árvore binária balanceada, mas não necessariamente completa, i.e., os irmãos do último nó interno tem exatamente dois filhos, e este pode ter um único filho esquerdo. Adicionalmente, um *heap* satisfaz a seguinte relação de ordem: a chave de cada nó interno é maior ou igual que as chaves dos seus filhos ([CLRS01, BvG99]). Por exemplo, a árvore:



é um *heap* e pode ser representado como a sequência finita:

9	8	4	5	6	3	0	0	2	1

A função `heapify` deve transformar sequências finitas em um heap, e está especificada como:

```
heapify(h)(i : nat | i <= floor(length(h)/2) - 1) :
RECURSIVE finite_sequence[nat] =
IF i > 0 THEN
heapify(sink(h)(i, length(h) - 1))(i - 1)
ELSE
sink(h)(i, length(h) -1)
ENDIF
MEASURE i
```

Logo `heapsort` aplica a função recursiva `heapsort_aux`, abaixo, utilizando como parâmetros o resultado de `heapify` e o comprimento do heap, que se encarrega de todo o processo de ordenação.

```
heapsort_aux(h)(n :below[length(h)]): RECURSIVE finite_sequence[nat] =
IF n = 0 THEN h
ELSIF n = 1 THEN swap(h)(0,1)
ELSE heapsort_aux(sink(swap(h)(0,n))(0, n - 1))(n - 1)
ENDIF
MEASURE n
```

`heapsort_aux` aproveita a ordenação do `heap`, que implica que a raiz da estrutura contém a maior chave, colocando-o então na última posição, por meio da função `swap`.

```
swap(h)(i, j : below[length(h)]) :  finite_sequence[nat] =
(# length := length(h) ,
seq := (LAMBDA (k : below[length(h)]) :
IF k = i THEN h(j)
ELSIF k = j THEN h(i)
ELSE h(k) ENDIF)  #)
```

Depois disto, `heapsort` reorganiza o `heap` por meio da função `sink` até a posição anterior àquela da troca do maior elemento. O processo se realiza recursivamente:

```
sink(h)((i : below[length(h)]), (n : below[length(h)] | n >= i)) :
RECURSIVE finite_sequence[nat] =
IF i > floor((n+1)/2) - 1 THEN h
ELSE LET k = ind_gc(h)(n,i) IN
IF h(k) > h(i) THEN sink(swap(h)(i,k))(k,n)
ELSE h ENDIF
ENDIF
MEASURE n - i
```

3 Questões

Deverão ser formalizados resultados a seguir, conforme a especificação nas teorias disponibilizadas na especificação do projeto: `sorting_seq.pvs` e `heapsort.pvs`.

Questão 01 - desafio *Permutações tem o mesmo comprimento.* A primeira questão do projeto está relacionada com a correspondência entre especificações de occorrências, e permutações sobre listas e sequências finitas. Especificamente, deseja-se formalizar indutivamente, que sequências finitas que são permutações tem o mesmo comprimento.

```
fsq_permutations_length_by_induction : CONJECTURE
    FORALL(h1,h2:finite_sequence[T]) :
        permutations(h1,h2) => length(h1) = length(h2)
```

Esse resultado já está formalizado como o corolário `fsq_permutations_length`, mas o que se quer nesta questão é uma prova direita indutiva, sem utilização dos lemas aplicados na formalização do corolário (veja a prova do corolário).

Questão 02 *Formalizar que o resultado de aplicar a função* `sink` *sobre um heap preserva o comprimento do heap.*

```
sink_length : COROLLARY
    FORALL (h,  (i : nat),  (n : nat |  n < length(h) AND n >= i)) :
                    length(sink(h)(i,n)) = length(h)
```

Questão 03

Formalizar que o resultado de aplicar `heapify` *num heap preserva o comprimento do heap*

```
heapify_length : COROLLARY
     FORALL (h, (i: below[floor(length(h)/2)])) :
                         length(heapify(h)(i)) = length(h)
```

Questão 04

Formalizar que o resultado de aplicar a função `heapsort_aux` *preserva o comprimento do heap.*

```
heapsort_aux_length : LEMMA
    FORALL (h, (n: below[length(h)])) :
               length(heapsort_aux(h)(n)) = length(h)
```

4 Etapas do desenvolvimento do projeto

O projeto será dividido em duas etapas como segue:

- A primeira etapa do projeto e a de Verificação das Formalizações. Os grupos deverão ter prontas as suas formalizações na linguagem do assistente de demonstração PVS e baixar cópia dos arquivos comprimidos do projeto (arquivos (`sortin_seq.pvs`, `sorting_seq.prf`), `heapsort.pvs` e `heapsort.prf`) até o dia **22.08.2022, 8am**. Na mesma semana dias **22 e 24.08.2022**, durante o horário de aula no LaForCE, Laboratório do Grupo de Teoria da Computação da UnB, alocado no prédio do CIC, realizar-se-á a verificação do trabalho para a qual os membros dos grupos deverão comparecer integralmente, segundo horário determinado para cada grupo (40 minutos).
  Avaliação (peso 6.0):
  - Um dos membros, selecionado por sorteio, explicará os detalhes da formalização em, no máximo, 20 minutos.
  - Os quatro membros do grupo poderão então complementar a explicação inicial em, no máximo, 10 minutos.
  - A formalização será testada durante a verificação.

  O objetivo desta verificação é determinar se os elementos dedutivos teóricos foram corretamente assimilados para aplicação coerente e eficiente dos comandos de prova do assistente de demonstração utilizado no projeto.

- A segunda etapa do projeto consiste da apresentação dos resultados finais e conclusões do estudo do problema.
  Avaliação (peso 4.0): Cada grupo de trabalho devera entregar um Relatório Final inédito, editado em LaTeX, limitado a oito páginas (12 pts, A4, espaçamento simples) do projeto até o dia **29.08.2022, 8am** via plataforma Aprender3, com o seguinte conteúdo:

- Introdução e contextualização do problema.
  Nota: o problema genericamente consiste em verificação da aplicabilidade de dedução formal para verificação de algoritmos.
- Explicação das técnicas de solução.
  Nota: as técnicas de solução estão relacionadas principalmente aos mecanismos dedutivos lógicos aplicados, e em segundo lugar, e muito menos relevante, aos mecanismos de prova disponíveis no assistente de demonstração utilizado. Observa-se que existem diversos assistentes de demonstração que poderiam ser aplicados para realizar verificação de correção algorítmica.
- Especificação do problema e explicação do método de solução.
  Nota: focar nesta seção na técnica de ordenação heapsort e a problemática particular referente às demonstrações formalizadas.
- Descrição da formalização.
  Nota: sempre que como parte do projeto são entregues os documentos .pvs e .prf com especificações e formalizações, não se trata nesta seção de repetir os passos da formalização, mas de explicar e destacar de forma sucinta o que foi relevante nas formalizações.
- Conclusões.
  Nota: as conclusões boas precisam ser abrangentes e o mais genéricas possíveis; por exemplo, esperar-se-ia do exercício neste projeto que os alunos adquiram uma compreensão de que técnicas dedutivas formais lógicas podem ser aplicadas para verificação da correção de objetos computacionais (software e hardware). Conclusões ruins em geral não escalam aquém do exercício meramente realizado; por exemplo, uma conclusão péssima é algo do estilo "PVS serve para demonstrar correção de heapsort."Evite aqui conclusões ruins e restritas e realize um esforço para escalar o aprendizado da execução do projeto em outras áreas da computação e algorítmica.
- Referências.
  Nota: apresentação correta de referências é sempre um plus de relatórios experimentais. Esperar-se-ia, leitura de documentos ou textos explicando o algoritmo heapsort, documentos explicando dedução lógica, assistentes de demonstração, PVS, etc.

Referências

[AdM17] M. Ayala-Rincón and F. L. C. de Moura. *Applied Logic for Computer Scientists - Computational Deduction and Formal Proofs*. Undergraduate Topics in Computer Science. Springer, 2017.

[BvG99] S. Baase and A. van Gelder. *Computer Algorithms — Introduction to Design and Analysis*. Addison-Wesley, 1999.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Electrical Engineering and Computer Science Series. MIT press, second edition, 2001.